UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

AD-A093186

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AIM 554	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): EMACS MANUAL FOR ITS USERS	5. TYPE OF REPORT & PERIOD COVERED: Memorandum
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Richard M. Stallman	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C- 0643
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, Massachusetts 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 191
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, Virginia 22209	12. REPORT DATE: 8 June 1980
	13. NUMBER OF PAGES: 218
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, Virginia 22217	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

AI-M-554

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC ELECTE DEC 24 1980 S A

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Display Editor
Reference Manual
Primer

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This manual documents the use and simple customization of the display editor EMACS with the ITS operating system. The reader is not expected to be a programmer. Even simple customizations do not require programming skill, but the user who is not interested in customizing can ignore the scattered customization hints. This is primarily a reference manual, but can also be used a primer.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

4074 x 3

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 554

8 June 1980

EMACS Manual for ITS Users

by

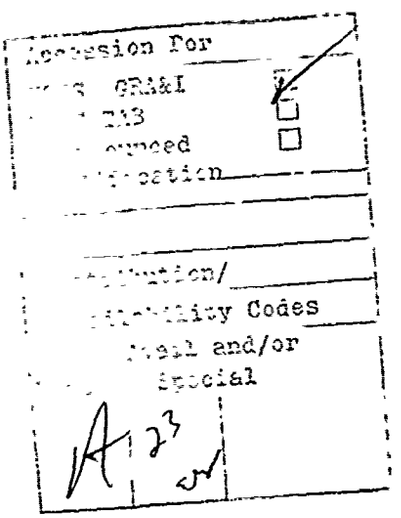
Richard M. Stallman

A reference manual

for the extensible, customizable, self-documenting

real-time display editor

This manual corresponds to EMACS version 147



This report describes work done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

Table of Contents

Introduction 3

1. The Organization of the Screen 5

1.1. The Mode Line 6

2. Character Sets and Command Input Conventions 9

2.1. The 9-bit Command Character Set 9
2.2. Prefix Characters 10
2.3. Commands, Functions, and Variables 10
2.4. Notational Conventions for ASCII Characters 11

3. Basic Editing Commands 13

3.1. Inserting Text 13
3.2. Moving The Cursor 13
3.3. Erasing Text 14
3.4. Files 14
3.5. Help 15
3.6. Using Blank Lines Can Make Editing Faster 15

4. Giving Numeric Arguments to EMACS Commands 17

4.1. Autoarg Mode 18

5. Extended (Meta-X) Commands and Functions 19

5.1. Issuing Extended Commands 19
5.2. Arcane Information about M-X Commands 21

6. Moving Up And Down Levels 25

6.1. Subsystems 25
6.2. Recursive Editing Levels 26
6.3. Exiting Levels; Exiting EMACS 26

7. Self-Documentation Commands 29

8. The Mark and the Region 31

8.1. Commands to Mark Textual Objects 32
8.2. The Ring of Marks 32

9. Killing and Moving Text 35

9.1. Deletion and Killing 35
9.2. Un-Killing 37
9.3. Other Ways of Copying Text 38

10. Searching 41

11. Commands for English Text 43

11.1. Word Commands 43
11.2. Sentence and Paragraph Commands 44
11.3. Indentation Commands for Text 46
11.4. Text Filling 47
11.5. Case Conversion Commands 49

11.6. Font-Changing 50
11.7. Underlining 51

12. Commands for Fixing Typos 53

12.1. Killing Your Mistakes 53
12.2. Transposition 53
12.3. Case Conversion 54

13. File Handling 55

13.1. Visiting Files 55
13.2. How to Undo Drastic Changes to a File 57
13.3. Auto Save Mode: Protection Against Crashes 57
13.4. Listing a File Directory 59
13.5. Cleaning a File Directory 59
13.6. DIRED, the Directory Editor Subsystem 60
13.7. Miscellaneous File Operations 63
13.8. The Directory Comparison Subsystem 64

14. Using Multiple Buffers 67

14.1. Creating and Selecting Buffers 67
14.2. Using Existing Buffers 68
14.3. Killing Buffers 68

15. Controlling the Display 71

16. Two Window Mode 73

16.1. Multiple Windows and Multiple Buffers 74

17. Narrowing 77

18. Commands for Manipulating Pages 79

18.1. Editing Only One Page at a Time 80

19. Replacement Commands 83

19.1. Query Replace 83
19.2. Other Search-and-loop Functions 84
19.3. TECO Search Strings 85

20. Editing Programs 87

20.1. Major Modes 87
20.2. Compiling Your Program 88
20.3. Indentation Commands for Code 89
20.4. Automatic Display Of Matching Parentheses 90
20.5. Manipulating Comments 91
20.6. Lisp Mode and Muddle Mode 93
20.7. Lisp Grinding 96
20.8. Editing Assembly-Language Programs 97
20.9. Major Modes for Other Languages 98

21. The TAGS Package. 101

21.1. How to Make a Tags File for a Program 101
21.2. How to Tell EMACS You Want to Use TAGS 102
21.3. Jumping to a Tag 103
21.4. Other Operations on Tag Tables 103

21.5. What Constitutes a Tag 105
21.6. Adding or Removing Source Files 106
21.7. How a Tag Is Described in the Tag Table 107
21.8. Tag Tables for INFO Structured Documentation Files 108

22. Simple Customization 111

22.1. Minor Modes 111
22.2. Libraries of Commands 112
22.3. Variables 114
22.4. The Syntax Table 115
22.5. FS Flags 117
22.6. Local Variables in Files 118
22.7. Init Files and EVARS Files 120
22.8. Keyboard Macros 124

23. The Minibuffer 127

24. Correcting Mistakes and EMACS Problems 129

24.1. Cancelling a Command 129
24.2. What to Do if EMACS Acts Strangely 130
24.3. Undoing Changes to the Buffer 132
24.4. Journal Files 133
24.5. Reporting Bugs 136

25. Word Abbreviation Input 141

25.1. Basic Usage 142
25.2. Advanced Usage 145
25.3. Teco Details for Extension Writers 148

26. The PICTURE Subsystem, an Editor for Text Pictures 151

27. Sorting Functions 153

Appendix I. Particular Types of Terminals 155

I.1. Ideal Keyboards 155
I.2. Keyboards with an "Edit" key 156
I.3. ASCII Keyboards 156
I.4. Upper-case-only Terminals 157
I.5. The SLOWLY Package for Slow Terminals 158

Appendix II. Use of EMACS from Printing Terminals 161

Glossary 163

Command Index 171

Catalog of Libraries 185

Index of Variables 189

EMACS Command Chart (as of 03/27/80) 195

Index 203

Preface

This manual documents the use and simple customization of the display editor EMACS with the ITS operating system. The reader is not expected to be a programmer. Even simple customizations do not require programming skill, but the user who is not interested in customizing can ignore the scattered customization hints.

This is primarily a reference manual, but can also be used as a primer. However, I recommend that the newcomer first use the on-line, learn-by-doing tutorial TEACHEMACS, by typing :TEACHEMACS<cr> while in HACTRN. With it, you learn EMACS by using EMACS on a specially designed file which describes commands, tells you when to try them, and then explains the results you see. This gives a more vivid introduction than a printed manual.

On first reading, you need not make any attempt to memorize chapters 1 and 2, which describe the notational conventions of the manual and the general appearance of the EMACS display screen. It is enough to be aware of what questions are answered in these chapters, so you can refer back when you later become interested in the answers. After reading the Basic Editing chapter you should practice the commands there. The next few chapters describe fundamental techniques and concepts which are referred to again and again. It is best to understand them thoroughly, experimenting with them if necessary.

To find the documentation on a particular command, look in the index if you know what the command is. If you know vaguely what the command does, look in the command index. The command index contains a line or two about each command, and a cross-reference to the section of the manual which describes the command in more detail. Related commands are grouped together. There is also a glossary, with a cross reference for each term.

The manual is available in three forms: the published form, the LPT form, and the INFO form. The published form is printed by the Artificial Intelligence lab. The LPT form is available on line for printing on unsophisticated hard copy devices such as terminals and line printers. The INFO form is for on-line perusal with the INFO program. All three forms are substantially the same. There are also two versions of the text: one for use with ITS, and one for use with Twenex, DEC's "TOPS-20" system. Both versions are available in all three forms.

EMACS is available for distribution for use on Tenex and Twenex systems (It does not run on Bottoms-10, and the conversion would not be easy). Mail us a 2400 foot mag tape if you want it. It does not cost anything; instead, you must join the EMACS software-sharing commune. The conditions of membership are that you must send back any improvements you make to EMACS, including any libraries you write, and that you must not redistribute the system except exactly as you got it, complete. (You can also distribute your

customizations, *separately*.) It is pathetic to hear from sites which received incomplete copies lacking the sources, asking me years later whether sources are available.

For information on the underlying philosophy of EMACS and the lessons learned from its development, write to me for a copy of AI memo 519, "EMACS, the Extensible, Customizable Self-Documenting Display Editor", or send Arpanet mail to RMS@MIT-AI

Yours in hacking,

```
 / 2 \ 1/2
< X  >
 \   /
```

Richard M. Stallman
MIT Artificial Intelligence Lab
545 Tech Square, rm 913
Cambridge, MA 02139
(617) 253-6765

Introduction

You are about to read about EMACS, an advanced, self-documenting, customizable, extensible real-time display editor.

We say that EMACS is a display editor because normally the text being edited is visible on the screen and is updated automatically as you type your commands. See section 1 [Display], page 5.

We call it a real-time editor because the display is updated very frequently, usually after each character or pair of characters the user types. This minimizes the amount of information you must keep in your head as you edits. See section 3 [Basic], page 13.

We call EMACS advanced because it provides facilities that go beyond simple insertion and deletion: filling of text; automatic indentation of programs; viewing two files at once; and dealing in terms of characters, words, lines, sentences, paragraphs, and pages, as well as expressions and comments in several different programming languages. It is much easier to type one command meaning "go to the end of the paragraph" than to find the desired spot with repetition of simpler commands.

Self-documenting means that at any time you can type a special character, the "Help" key, to find out what your options are. You can also use it to find out what any command does, or to find all the commands that pertain to a topic. See section 7 [Help], page 29.

Customizable means that you can change the definitions of EMACS commands in little ways. For example, if you use a programming language in which comments start with <** and end with **>, you can tell the EMACS comment manipulation commands to use those strings. Another sort of customization is rearrangement of the command set. For example, if you prefer the four basic cursor motion commands (up, down, left and right) on keys in a diamond pattern on the keyboard, you can have it. See section 22.1 [MinorModes], page 111.

Extensible means that you can go beyond simple customization and write entirely new commands, programs in the language TECO. EMACS is an "online extensible" system, which means that it is divided into many functions which call each other, any of which can be redefined in the middle of an editing session. Any part of EMACS can be replaced without making a separate copy of all of EMACS. Many already written extensions are distributed with EMACS, and some (including DIRED, PAGE, PICTURE, SORT, TAGS, and WORDAB) are documented in this manual. Although only a programmer can write an extension, anybody can use it afterward.

Extension requires programming in TECO, a rather obscure language. If you are clever and bold, you

might wish to learn how. See the file INFO;CONV >, for advice on learning TECO. This manual does not even try to explain how to write TECO programs, but it does contain some notes which are useful primarily to the extension writer.

1. The Organization of the Screen

EMACS divides the screen into several areas, each of which contains its own sorts of information. The biggest area, of course, is the one in which you usually see the text you are editing. The terminal's cursor usually appears in the middle of the text, showing the position of "point", the location at which editing takes place. While the cursor appears to point *at* a character, point should be thought of as *between* two characters: it points *before* the character that the cursor appears on top of. Terminals have only one cursor, and when output is in progress it must appear where the typing is being done. This does not mean that point is moving. It is only that EMACS has no way to show you the location of point except when the terminal is idle.

The C-X = command tells you precisely what is in the text, if it is not clear from the display. (If you are a beginner, don't worry if you don't understand this paragraph). It prints the row and column of the location of the cursor on the screen, and the numeric code for the character after the cursor. See section 11.4 [Filling], page 47.

The top lines of the screen are usually available for text but are sometimes pre-empted by an "error message", which says that some command you gave was illegal or used improperly, or by typeout from a command (such as, a listing of a file directory). The error message or typeout appears there for your information, but it is not part of the file you are editing, and it goes away if you type any command. If you want to make it go away immediately but not do anything else, you can type a Space. (Usually a Space inserts itself, but when there is an error message or typeout on the screen it does nothing but get rid of that.) The terminal's cursor always appears at the end of the error message or typeout, but this does not mean that point has moved. The cursor moves back to the location of point after the error message or typeout goes away.

A few lines at the bottom of the screen compose what is called "the echo area". "Echoing" means printing out the commands that you type. EMACS commands are usually not echoed at all, but if you pause for more than a second in the middle of a multi-character command then the whole command (including what you have typed so far) is echoed. This behavior is designed to give confident users optimum response, while giving nervous users information on what they are doing.

EMACS also uses the echo area for reading and displaying the arguments for some commands, such as searches, and for printing information in response to certain commands.

The line above the echo area is known as the "mode line". It is the line that usually starts with "EMACS (something)". Its purpose is to tell what is going on in the EMACS, and to show any reasons why commands may not be interpreted in the standard way. The mode line is very important, and if you are surprised by how EMACS reacts to your commands you should look there for enlightenment.

1.1. The Mode Line

The normal situation is that characters you type are interpreted as EMACS commands. When this is so, you are at "top level", and the mode line has this format:

```
EMACS type  (major minor)  bfr:  file --pos-- *
```

"type" is usually not there. When it is there, it indicates that the EMACS job you are using is not an ordinary one, in that it is acting as the servant of some other job. A type of "LEDIT" indicates an EMACS serving one or more Lisps, while a type of "MAIL" indicates an EMACS which you got by asking for an "edit escape" while composing mail to send. The type can also indicate a subsystem which is running, such as RMAIL. The type is stored internally as a string in the variable Editor Type. The variable is normally zero.

"major" is always the name of the "major mode" you are in. At any time, EMACS is in one and only one of its possible major modes. The major modes available include Fundamental mode, Text mode (which EMACS starts out in), Lisp mode, PASCAL mode, and others. See section 20 [Major Modes], page 87, for details of how the modes differ and how to select one. Sometimes the name of the major mode is followed immediately with another name inside square-brackets ("[-]"). This name is called the "submode". The submode indicates that you are "inside" of a command which causes your editing commands to be changed temporarily, but does not change *what* text you are editing. A submode is a kind of recursive editing level. See section 6.2 [Submodes], page 26.

"minor" is a list of some of the minor modes which are turned on at the moment. "Fill" means that Auto Fill mode is on. "Save" means that Auto-saving is on. "Save(off)" means that Auto-saving is on in general but momentarily turned off (it was overridden the last time a file was selected). "Atom" means that Atom Word mode is on. "Abbrev" means that Word Abbrev mode is on. "Ovwt" means that Overwrite mode is on. See section 22.1 [Minor Modes], page 111, for more information. "Def" means that a keyboard macro is being defined; although this is not exactly a minor mode, it is still useful to be reminded about. See section 22.8 [Keyboard Macros], page 124.

"bfr" is the name of the currently selected buffer. Each buffer has its own name and holds a file being edited; this is how EMACS can hold several files at once. But at any time you are editing only one of them, the "selected" buffer. When we speak of what some command does to "the buffer", we are talking about the currently selected buffer. Multiple buffers makes it easy to switch around between several files, and then it is very useful that the mode line tells you which one you are editing at any time. However, before you learn how to use multiple buffers, you will always be in the buffer called "Main", which is the only one which exists when EMACS starts up. If the name of the buffer is the same as the first name of the file you are visiting, then the buffer name is left out of the mode line. See section 14 [Buffers], page 67, for how to use more than one buffer in one EMACS.

"file" is the name of the file that you are editing. It is the last file that was visited in the buffer you are in. If "(RO)" (for "read only") appears after the filename, it means that if you visit another file in the same buffer then changes you have made to this file will be lost unless you have explicitly asked to save them. See section 13.1 [Visiting], page 55, for more information. If there is no "(RO)" and you visit another file in the same buffer, EMACS will offer to save your changes first, if there are any changes.

The star at the end of the mode line means that there are changes in the buffer which have not been saved in the file. If the file has not been changed since it was read in or saved, there is no star.

"pos" tells you whether there is additional text above the top of the screen, or below the bottom. If your file is small and it is all on the screen, --pos-- is omitted. Otherwise, it is --TOP-- if you are looking at the beginning of the file, --BOT-- if you are looking at the end of the file, or --nn%-- where nn is the percentage of the file above the top of the screen.

Sometimes you will see --MORE-- instead of --nn%--. This happens when typeout from a command is too long to fit on the screen. It means that if you type a Space the next screenful of information will be printed. If you are not interested, typing anything but a Space will cause the rest of the output to be discarded. Typing a Rubout will discard the output and do nothing else. Typing any other command will discard the rest of the output and also do the command. When the output is discarded, "FLUSHED" is printed after the --MORE--.

So much for what the mode line says at top level. When the mode line doesn't start with "EMACS", and doesn't look anything like the breakdown given above, then EMACS is not at top level, and your typing will not be understood in the usual way. This is because you are inside a subsystem, such as INFO (See section 6.1 [Subsystems], page 25.), or in a recursive editing level, such as Edit Options (See section 6.2 [Recursive Editing], page 26.). The mode line tells you what command you are inside.

If you are accustomed to other display editors, you may be surprised that EMACS does not always display the page number and line number of point in the mode line. This is because the text is stored in a way which makes it difficult to compute this information. Displaying them all the time would be too slow to be borne. When you want to know the page and line number of point, you must ask for the information with the M-X What Page command. See section 18 [Pages], page 79. However, once you are adjusted to EMACS, you will rarely have any reason to be concerned with page numbers or line numbers.

2. Character Sets and Command Input Conventions

In this chapter we introduce the terminology and concepts used to talk about EMACS commands. In particular, EMACS is designed for a kind of keyboard with two special shift keys which can type 512 different characters, instead of the 128 which ordinary ASCII keyboards can send.

2.1. The 9-bit Command Character Set

EMACS is designed ideally to be used with terminals whose keyboards have a pair of shift keys, labelled "Control" and "Meta", either or both of which can be combined with any character that you can type. These shift keys produce "Control" characters and "Meta" characters, which are the editing commands of EMACS. Ordinary characters like "A" which are neither Control nor Meta are used for inserting text. We name each of these characters by prefixing "Control-" or "Meta-" (abbreviated "C-" and "M-") to the character: thus, Meta-F or M-F is the character which is F typed with the Meta key held down. Control in the EMACS command character set is not precisely the same as Control in the ASCII character set, but the general purpose is the same.

The 128 characters, multiplied by the four possibilities of the Control and Meta keys, make 512 characters in the EMACS command character set. So it is called the 512-character set to distinguish it from ASCII, which has only 128 characters. It is also called the "9-bit" character set because 9 bits are required to express a number from 0 to 511. Note that the 512-character set is used only for keyboard commands. Characters in files being edited with EMACS are ASCII characters.

Sadly, most terminals do not have ideal EMACS keyboards. In fact, the only ideal keyboards are at MIT. On nonideal keyboards, the Control key is somewhat limited (it can be combined with only some other characters, not with all), and the Meta key may not exist at all. We make it possible to use EMACS on a nonideal terminal by providing two-character circumlocutions, made up of characters that you can type, for the characters that you can't type. These circumlocutions start with a "bit prefix character"; see below. Also see the appendix for more detailed information on what to do on your type of terminal.

It may seem an unfortunate coincidence that both the EMACS 9-bit character set and the ASCII character set use the term "Control" for not exactly the same thing. This came about because the 9-bit character set was invented by generalizing ASCII. In ASCII, only letters and a few punctuation marks can be made into Control characters; we wanted to have a Control version of every character. For example, we have Control-Space, Control-digits, and Control-=. We also have Control-A and Control-a which are two different characters; however, all such pairs have the same meaning as EMACS commands, so you can forget about this quirk of the character set unless you begin customizing. In practice, you can forget all about the

distinction between ASCII Control and EMACS Control, except to realize that EMACS uses some "Control" characters which are not on your keyboard.

In addition to the 9-bit command character set, there is one extra character called Help. It cannot be combined with Control or Meta. Its use is to ask for documentation, at any time. Like the the 9-bit characters, the Help character has its own key on an ideal keyboard, but must be represented by something else on other keyboards. The circumlocution we use is Control-_ H (two characters). The code used internally for Help is 4110 (octal).

We have given some command characters special names which we always capitalize. "Return" or "<cr>" stands for the carriage return character, code 015 (all character codes are in octal). Note that C-R means the character Control-R, never Return. "Rubout" is the character with code 177, labeled "Delete" on some keyboards. "Altmode" is the character with code 033, sometimes labeled "Escape". Other command characters with special names are Tab (code 011), Backspace (code 010), Linefeed (code 012), Space (code 040), Excl ("!", code 041), Comma (code 054), and Period (code 056). Control is represented in the numeric code for a character by 200, and Meta by 400; thus, Meta-Period is code 456 in the 9-bit character set.

2.2. Prefix Characters

A non-ideal keyboard can only send certain Control characters, and may completely lack the ability to send Meta characters. To use these commands on such keyboards, you need to use two-character circumlocutions starting with a "bit prefix" character which turns on the Control or Meta bit in the second character. The Altmode character turns on the Meta bit, so Altmode X can be used to type a Meta-X, and Altmode Control-O can be used to type a C-M-O. Altmode is known as "the Metizer". Other bit prefix characters are C-^ for Control, and C-C for Control and Meta together. Thus, C-^ < is a way of typing a Control-<, and C-C < can be used to type C-M-<. Because C-^ is awkward to type on most keyboards, we have tried to minimize the number of commands for which you will need it.

There is another prefix character, Control-X which is used as the beginning of a large set of two-character commands known as "C-X commands". C-X is not a bit prefix character, C-X A is not a circumlocution for any single character, and it must be typed as two characters on any terminal.

2.3. Commands, Functions, and Variables

Most of the EMACS commands documented herein are members of this 9-bit character set. Others are pairs of characters from that set. However, EMACS doesn't really define commands directly. Instead, EMACS defines "functions", which have long names such as "^R Down Real Line", and then the functions are connected to "commands" such as C-N through a dispatch table. When we say that C-N moves the cursor

down a line we are glossing over a distinction which is unimportant for ordinary use, but essential for customization: it is the function ^R Down Real Line which knows how to move down a line, and C-N moves down a line *because* it is connected to that function. We usually ignore this subtlety to keep things simple. To give the extension-writer the information he needs, we state the name of the function which really does the work in parentheses after mentioning the command name. For example: "C-N (^R Down Real Line) moves the cursor down a line". In the EMACS wall chart, the function names are used as a form of very brief documentation for the command characters. See section 5.2 [Functions], page 21.

The "^R " which appears at the front of the function name is simply part of the name. By convention, a certain class of functions have names which start with "^R ".

While we are on the subject of customization information which you should not be frightened of, it's a good time to tell you about variables. Often the description of a command will say "to change this, set the variable Mumble Foo". A variable is a name used to remember a value. EMACS contains many variables which are there so that you can change them if you want to customize. The variable's value is examined by some command, and changing the value makes the command behave differently. Until you are interested in customizing, you can ignore this information. When you are ready to be interested, read the basic information on variables, and then the information on individual variables will make sense. See section 22.3 [Variables], page 114.

2.4. Notational Conventions for ASCII Characters

Control characters in files, your EMACS buffer, or TECO programs, are ordinary ASCII characters and are represented as uparrow or caret followed by the corresponding non-control character: control-E is represented as ↑E. The special 9-bit character set applies only to typing EMACS commands.

CRLF is the traditional term for a carriage return followed by a linefeed. This sequence of two characters is what separates lines in text being edited. Normally, EMACS commands make this sequence appear to be a single character, but TECO code must deal with the realities. A return or a linefeed which is not part of a CRLF is called "stray". EMACS usually treats them as part of the line and displays them as ↑M and ↑J.

Other ASCII characters with special names include tab (control-I, 011), backspace (control-H, 010), linefeed (control-J, 012), altmode (033), space (040), and rubout (177). Tab and control-I are different as 9-bit command characters, but when reduced to ASCII they become the same. Our convention is that names of ASCII characters are in lower case, while names of 9 bit command characters are in upper case.

Most control characters when present in the EMACS buffer are displayed with a caret; thus, ^A for ASCII ↑A. Rubout is displayed as ^?, because by stretching the meaning of "control" it can be interpreted as ASCII

control-?. A backspace is usually displayed as ^H since it is ASCII control-H, because most displays cannot do overprinting.

Altmode is the ASCII code 033 sometimes labeled "Escape" or "Alt". Altmode is often represented by itself in this document (remember, it is an ASCII character and can therefore appear in files). It looks like this: ♦. On some terminals, altmode looks just like the dollar sign character. If that's so on yours, you should assume that anything you see in the on-line documentation which looks like a dollar sign is really an altmode unless you are specifically told it's a dollar sign. The dollar sign character is not particularly important in EMACS and we will rarely have reason to mention it.

3. Basic Editing Commands

We now give the basics of how to enter text, make corrections, and save the text in a file. If this material is new to you, you might learn it more easily by running the TEACHEMACS program.

3.1. Inserting Text

To insert printing characters into the text you are editing, just type them. Normally (when EMACS is at top level), they are inserted into the text at the cursor, which moves forward. Any characters after the cursor move forward too. If the cursor is in between a FOO and a BAR, typing XX produces and displays FOOXXBAR with the cursor before the "B". This method of insertion works for printing characters and space, but other characters act as editing commands and do not insert themselves. If you need to insert a control character, Altmode, Tab or Rubout, you must quote it by typing the C-Q command first. "C" refers to the Control bit. See section 2 [Characters], page 9.

To correct text you have just inserted, you can use Rubout. Rubout deletes the character *before* the cursor (not the one that the cursor is on top of or under; that is the character *after* the cursor). The cursor and all characters after it move backwards. You can rub out a line boundary by typing Rubout when the cursor is at the beginning of a line.

To end a line and start typing a new one, type Return (^R CRLF). You can also type Return to break an existing line into two. A Rubout after a Return will undo it. Return really inserts two characters, a carriage return and a linefeed (a CRLF), but almost everything in EMACS makes them look like just one character, which you can think of as a line-separator character.

If you add too many characters to one line, without breaking it with a Return, the line will grow to occupy two (or more) lines on the screen, with a "!" at the extreme right margin of all but the last of them. The "!" says that the following screen line is not really a distinct line in the file, but just the "continuation" of a line too long to fit the screen.

3.2. Moving The Cursor

To do more than insert characters, you have to know how to move the cursor. Here are a few of the commands for doing that.

C-A Moves to the beginning of the line.
C-E Moves to the end of the line.
C-F Moves forward over one character.

C-B Moves backward over one character.
C-N Moves down one line, vertically. If you start in the middle of one line, you end in the middle of the next. From the last line of text, it creates a new line.
C-P Moves up one line, vertically.
C-L Clears the screen and reprints everything. C-U C-L reprints just the line that the cursor is on.
C-T Transposes two characters (the ones before and after the cursor).
M-< Moves to the beginning of your text.
M-> Moves to the end of your text.

3.3. Erasing Text

Rubout Delete the character before the cursor.
C-D Delete the character after the cursor.
C-K Kill to the end of the line.

You already know about the Rubout command which deletes the character before the cursor. Another command, Control-D, deletes the character after the cursor, causing the rest of the text on the line to shift left. If Control-D is typed at the end of a line, that line and the next line are joined together.

To erase a larger amount of text, use the Control-K command, which kills a line at a time. If Control-K is done at the beginning or middle of a line, it kills all the text up to the end of the line. If Control-K is done at the end of a line, it joins that line and the next line.

See section 9.1 [Killing], page 35, for more flexible ways of killing text.

3.4. Files

The commands above are sufficient for creating text in the EMACS buffer. The more advanced EMACS commands just make things easier. But to keep any text permanently you must put it in a file. You do that by choosing a filename, such as FOO, and typing C-X C-V FOO<cr>. This "visits" the file FOO (actually, FOO > on your working directory) so that its contents appear on the screen for editing. You can make changes, and then "save" the file by typing C-X C-S. This makes the changes permanent and actually changes the file FOO. Until then, the changes are only inside your EMACS, and the file FOO is not really changed. If the file FOO doesn't exist, and you want to create it, visit it as if it did exist. When you save your text with C-X C-S the file will be created.

Of course, there is a lot more to learn about using files. See section 13 [Files], page 55.

3.5. Help

If you forget what a command does, you can find out with the Help character. The Help character is Top-H if you have a Top key, or Control-_ H (two characters!) otherwise. Type Help followed by C and the command you want to know about. Help can help you in other ways as well. See section 7 [Help], page 29.

3.6. Using Blank Lines Can Make Editing Faster

C-O	Insert one or more blank lines after the cursor.
C-X C-O	Delete all but one of many consecutive blank lines.

One thing you should know is that it is much more efficient to insert text at the end of a line than in the middle. So if you want to stick a new line before an existing one, it is better to make a blank line there first and then type the text into it, rather than inserting the new text at the beginning of the existing line and then insert a line separator. It is also clearer what is going on while you are in the middle.

To make a blank line, you can type Return and then C-B. But there is a single character for this: C-O (Customizers: this is the built-in function ^R Open Line). So, instead of typing FOO Return to insert a line containing FOO, type C-O FOO. If you want to insert many lines, you should type many C-O's at the beginning (or you can give C-O an argument to tell it how many blank lines to make. See section 4 [Arguments], page 17, for how). As you then insert lines of text, you will notice that Return behaves strangely: it "uses up" the blank lines instead of pushing them down. If you don't use up all the blank lines, you can type C-X C-O (the function ^R Delete Blank Lines) to get rid of all but one. C-X C-O on a blank line among many blank lines reduces them to one. C-X C-O on a nonblank line deletes any blank lines which follow.

4. Giving Numeric Arguments to EMACS Commands

Any EMACS command can be given a numeric argument. Some commands interpret the argument as a repetition count. For example, giving an argument of ten to the C-F command (move forward one character) moves forward ten characters. With these commands, no argument is equivalent to an argument of one.

Some commands care only about whether there is an argument, and not about its value; for example, the command M-Q (^R Fill Paragraph) with no arguments fills text, but with an argument justifies the text as well.

Some commands use the value of the argument, but do something peculiar when there is no argument. For example, the C-K (^R Kill Line) command with an argument <n> kills <n> lines and the line separators that follow them. But C-K with no argument is special; it kills the text up to the next line separator, or, if point is right at the end of the line, it kills the line separator itself. Thus, two C-K commands with no arguments can kill a nonblank line, just like C-K with an argument of one.

The fundamental way of specifying an argument is to use the C-U (^R Universal Argument) command followed by the optional minus sign and the digits. C-U followed by a non-digit other than a minus sign has the special meaning of "multiply by four". It multiplies the argument for the next command by four. Two such C-U's multiply it by sixteen. Thus, C-U C-U C-F moves forward sixteen characters. It is a good way to move forward "fast", since it moves about 1/4 of a line on most terminals. Other useful combinations are C-U C-N, C-U C-U C-N (move down a good fraction of a screen), C-U C-U C-O (make "a lot" of blank lines), and C-U C-K (kill four lines). With commands like M-Q that care whether there is an argument but not what the value is, C-U is a good way of saying "I want an argument".

A few commands treat a plain C-U differently from an ordinary argument. A few others may treat an argument of just a minus sign differently from an argument of -1. These unusual cases will be described when they come up; they are always for reasons of convenience of use.

There are other, terminal-dependent ways of specifying arguments. They have the same effect but may be easier to type. See the appendix. If your terminal has a numeric keypad which sends something recognizably different from the ordinary digits, it is possible to program EMACS to allow use of the numeric keypad for specifying arguments. The libraries VT52 and VT100 provide such a feature for those two types of terminals. See section 22.2 [Libraries], page 112.

4.1. Autoarg Mode

Users of ASCII keyboards may prefer to use Autoarg mode, in which an argument can be specified for most commands merely by typing the digits. Digits preceding an ordinary inserting character are themselves inserted, but digits preceding an Altmode or Control character serve as an argument to it and are not inserted. To turn on this mode, set the variable Autoarg Mode nonzero.

Autoargument digits echo at the bottom of the screen; the first nondigit causes them to be inserted or uses them as an argument. To insert some digits and nothing else, you must follow them with a Space and then rub it out. C-G cancels the digits, while Rubout inserts them all and then rubs out the last.

5. Extended (Meta-X) Commands and Functions

M-X Begin an extended command. Follow by command name and arguments.
C-M-X Begin an extended command. Follow by the command name only; the command will ask for any arguments.
C-X Altmode
Re-execute recent extended command.

While the most often useful EMACS commands are accessible via one or two characters, the less often used commands go by long names to make them easier to remember. They are known as "extended commands" because they extend the set of two-character commands. They are also called "M-X commands", because they all start with the character Meta-X (^R Extended Command). The M-X is followed by the command's name, actually the name of a function to be called. Terminate the name of the function with a Return (unless you are supplying string arguments; see below). For example, Meta-X Auto Fill Mode<cr> invokes the function Auto Fill Mode. This function when executed turns Auto Fill mode on or off.

We say that M-X Foo<cr> calls "the function FOO". When documenting the individual extended commands, we will call them "functions" to avoid confusion between them and the one or two character "commands". We will also use "M-X" as a title like "Mr." for functions, as in "use M-X Foo". The "extended command" is what you *type*, starting with M-X, and what the command *does* is call a function. The name that goes in the command is the name of the command and is also the name of the function, and both terms will be used.

5.1. Issuing Extended Commands

5.1.1. Typing The Command Name

When you type M-X, the cursor moves down to the echo area at the bottom of the screen. "M-X" is printed there, and when you type the command name it echoes there. This is known as "reading a line in the echo area". You can use Rubout to cancel one character of the command name, or C-U or C-D to cancel the entire command name. A C-G cancels the whole M-X, and so does a Rubout when the command name is empty. These editing characters apply to anything which reads a line in the echo area.

The string "M-X" which appears in the echo area is called a "prompt". The prompt always tells you what sort of argument is required and what it is going to be used for; "M-X" means that you are inside of the M-X command and should type the name of a function to be called.

5.1.2. Completion

You can abbreviate the name of the command, as long as the abbreviation is unambiguous. You can also use completion on the function name. This means that you type part of the command name, and EMACS visibly fills in the rest, or as much as can be determined from the part you have typed.

You request completion by typing an Altmode. For example, if you type M-X Au Altmode, the "Au" expands to "Auto " because all command names which start with "Au" continue with "to ". If you ask for completion when there are several alternatives for the next character, the bell rings and nothing else happens. Altmode is also the way to terminate the command name and begin the string arguments, but it only does this if the command name completes in full. In that case, an Altmode (♦) appears after the command name in the echo area. (If the command name does not complete in full, it is ambiguous, so it would be useless to type the arguments yet).

Space is another way to request completion, but it completes only one word. Successive Spaces complete one word each, until either there are multiple possibilities or the end of the name is reached. If the first word of a command is Edit, List, Kill, View or What, it is sufficient to type just the first letter and complete it with a Space. (This does not follow from the usual definition of completion, since the single letter is ambiguous; it is a special feature added because these words are so common).

Typing "?" in the middle of the command name prints a list of all the command names which begin with what you have typed so far. You can then go on typing the name.

5.1.3. Numeric Arguments and String Arguments

Some functions can use numeric prefix arguments. Simply give the Meta-X command an argument and Meta-X will pass it along to the function which it calls. The argument appears before the "M-X" in the prompt, as in "69 M-X", to remind you that the function you call will receive a numeric argument.

Some functions require "string arguments" or "suffix arguments". For those functions, the function name is terminated with a single Altmode, after which come the arguments, separated by Altmodes. After the last argument, type a Return to cause the function to be executed. For example, the function Describe prints the full documentation of a function (or a variable) whose name must be given as a string argument. An example of using it is Meta-X Describe♦Apropos<cr>, which prints the full description of the function named Apropos.

An alternate way of calling extended commands is with the command C-M-X (^R Instant Extended Command). It differs from plain M-X in that the function itself reads any string arguments. This can be useful if the string argument is a filename or a command name, because the function knows that and gives the

argument special treatment such as completion. However, there are compensating disadvantages. For one thing, since the function has already been invoked, you can't rub out from the arguments into the function name. For another, it is not possible to save the whole thing, function name and arguments, for you to recall with C-X Altmode (see below). So C-M-X saves *nothing* for C-X Altmode. The prompt for C-M-X is "C-M-X". You can override it with the variable Instant Command Prompt.

5.1.4. Repeating an Extended Command

The last few extended commands you have executed are saved and you can repeat them. We say that the extended command is saved, rather than that the function is saved, because the whole command, including arguments, is saved.

To re-execute a saved command, use the command C-X Altmode (^R Re-execute Minibuffer). It retypes the last extended command and ask for confirmation. With an argument, it repeats an earlier extended command; 2 means repeat the next to the last command, etc. You can also use the minibuffer to edit a previous extended command and re-execute it with changes (See section 23 [Minibuffer], page 127.).

Note: Extended commands and functions were once called "MM commands", but this term is obsolete. If you see it in any user documentation, please report it. Ordinary one or two character commands used to be known as "^R" commands, and the term may still be used in the online documentation of some functions; please report this also.

5.2. Arcane Information about M-X Commands

You can skip this section if you are not interested in customization, unless you want to know what is going on behind the scenes.

5.2.1. MM

Extended commands were once called "MM commands, because "MM" is a TECO expression which looks up a command name to find the associated program, and runs that program. Thus, the TECO expression

```
MM Apropos♦Word♦
```

means to run the Apropos command with the argument "word". You could type this expression into a minibuffer and get the same results as you would get from Meta-X Apropos♦Word<cr>. In fact, for the first year or so, EMACS had no Meta-X command, and that's what people did. See section 23 [Minibuffer], page 127, for information on the minibuffer.

"MM" actually tells TECO to call the subroutine in q-register "M". The first "M" means "call", and the second "M" says what to call. This subroutine takes a string argument which is the name of a function and looks it up. Calling a function is built into TECO, but looking up the name is not; it is implemented by the program TECO calls "M". That's why "MM" is called that and not "Run" or "F↑Q".

5.2.2. Arguments in TECO Code

Functions can use one or two "prefix arguments" or "numeric arguments". These are numbers (actually, TECO expressions) which go before the "MM". Meta-X can only give the MM command one argument. If you want to give it two, you must type it in using the minibuffer. When TECO code passes prefix arguments, they don't have to be numbers; they can also be strings, TECO buffer objects, etc. However, no more about that here.

When TECO code passes a string argument, it appears terminated by an Altmode after the Altmode which ends the function name. There can be any number of string arguments. In fact, the function can decide at run time how many string arguments to read. This makes it impossible to compile TECO code!

5.2.3. Commands and Functions

Actually, *every* command in EMACS simply runs a function. For example, when you type the command C-N, it runs the function "^R Down Real Line". You could just as well do C-U 1 M-X ^R Down Real Line<cr> and get the same effect. C-N can be thought of as a sort of abbreviation. We say that the command C-N has been "connected" to the function ^R Down Real Line. The name is looked up once when the command and function are connected, so that it does not have to be looked up again each time the command is used. For historical reasons, the default argument passed to a function which is connected to a command you typed is 1, but the default for MM and for M-X is 0. This is why the C-U 1 was necessary in the example above. The documentation for individual EMACS commands usually gives the name of the function which really implements the command in parentheses after the command itself.

Just as any function can be called directly with M-X, so almost any function can be connected to a command. This is the basis of customization of EMACS. You can use the function Set Key to do this. To define C-N, you could type M-X Set Key◆^R Down Real Line<cr>, and then type C-N. If you use the function View File often, you could connect it to the command C-X V (not normally defined). You could even connect it to the command C-M-V, replacing that command's normal definition. This can be done with the function Set Key; or you can use an init file to do it permanently. See section 22.7 [Init], page 120.

5.2.4. Subroutines and Built-in Functions

EMACS is composed of a large number of functions, each with a name. Some of these functions are connected to commands; some are there for you to call with M-X; some are called by other functions. The last group are called subroutines. They usually have names starting with "&", as in "& Read Line", the subroutine which reads a line in the echo area. Although most subroutines have such names, any function can be called as a subroutine. Functions like ^R Down Real Line have names starting with ^R because you are not expected to call them directly, either. The purpose of the "&" or "^R" is to get those function names out of the way of command completion in M-X. M-X allows the command name to be abbreviated if the abbreviation is unique, and the commands that you are not interested in might have names that would interfere and make some useful abbreviation cease to be unique. The funny characters at the front of the name prevent this from happening.

Some function names, present as definitions of single-character commands, are known to all the Help features but don't seem to exist if you try to call them by name. The names of these functions are not always defined; they are contained in a library called BARE which is loaded temporarily by each documentation command and then flushed again. The reason for this is that these functions are really built into TECO and not part of EMACS; the EMACS "definitions" aren't necessary for actually using them, and are only there for the sake of documentation. If you load BARE permanently, then you can refer to these functions by name like all others. See section 22.2 [Libraries], page 112.

6. Moving Up And Down Levels

Subsystems and recursive editing levels are two states in which you are temporarily doing something other than editing the visited file as usual. For example, you might be editing a message that you wish to send, or looking at a documentation file with INFO.

6.1. Subsystems

A subsystem is an EMACS function which is an interactive program in its own right: it reads commands in a language of its own, and displays the results. You enter a subsystem by typing an EMACS command which invokes it. Once entered, the subsystem runs until a specific command to exit the subsystem is typed. An example of an EMACS subsystem is INFO, the documentation reading program. Others are Backtrace and TDEBUG, used for debugging TECO programs, and RMAIL and BABYL, used for reading and editing mail files.

The commands understood by a subsystem are usually not like EMACS commands, because their purpose is something other than editing text. For example, INFO commands are designed for moving around in a tree-structured documentation file. In EMACS, most commands are Control or Meta characters because printing characters insert themselves. In most subsystems, there is no insertion of text, so non-Control non-Meta characters can be the commands.

While you are inside a subsystem, the mode line usually gives the name of the subsystem (as well as other information supplied by the subsystem, such as the filename and node name in INFO). You can tell that you are inside a subsystem because the mode line does not start with "EMACS", or with an open bracket ("[") which would indicate a recursive editing level. See section 1.1 [Mode Line], page 6.

Because each subsystem implements its own commands, we cannot guarantee anything about them. However, there are conventions for what certain commands ought to do:

C-]	aborts (exits without finishing up)
Backspace	Scrolls backward, like M-V in EMACS.
Space	Scrolls forward, like C-V in EMACS.
Q	Exits normally.
X	Begins an extended command, like M-X in EMACS.
Help or ?	Prints documentation on the subsystem's commands.

Not all of these necessarily exist in every subsystem, however.

6.2. Recursive Editing Levels

A recursive editing level is a state in which you are inside a command which has given you some text for you to edit. The text may or may not be part of the file you are editing. Recursive editing levels are indicated in the mode line by square brackets ("[" and "]").

For example, the command M-X Edit Options is for changing the settings of EMACS options by editing a list of option names and values. You use the same commands as always for making changes in this list; when you are finished, the changes take affect in your option settings. While you are editing the list of options, the mode line says "[Edit Options]". See section 22.3 [Variables], page 114.

A recursive editing level differs from a subsystem in that the commands are ordinary EMACS commands (though a handful may have been changed slightly), whereas a subsystem defines its own command language.

The text you edit inside a recursive editing level depends on the command which invoked the recursive editing level. It could be a list of options and values, or a list of tab stop settings, syntax table settings, a message to be sent, or any text that you might wish to compose.

Sometimes in a recursive editing level you edit text of the file you are visiting, just as at top level. Why would this be? Usually because a few commands are temporarily changed. For example, Edit Picture in the PICTURE library defines commands good for editing a picture made out of characters, then enters a recursive editing level. When you exit, the special picture-editing commands go away. Until then, the brackets in the mode line serve to remind you that, although the text you are editing is your file, all is not normal. See section 26 [PICTURE], page 151.

In any case, if the mode line says "[...]" you are inside a recursive editing level, and the way to exit (send the message, redefine the options, get rid of the picture-editing commands, etc.) is with the command Control-Altmode or C-M-C (^R Exit). See section 6.3 [Exiting], page 26. If you change your mind about the command (you don't want to send the message, or change your options, etc.) then you should use the command C-] (Abort Recursive Edit) to get out. See section 24.1 [Aborting], page 129.

When the text in the mode line is surrounded by parentheses, it means that you are inside a "Minibuffer". A minibuffer is a special case of the recursive editing level. Like any other, it can be aborted safely with C-]. For full details on minibuffers, See section 23 [Minibuffer], page 127.

6.3. Exiting Levels; Exiting EMACS

C-X C-C	Exit from EMACS to the superior job.
C-M-C	Exit from EMACS or from a recursive editing level.

The general EMACS command to exit is C-M-C (^R Exit). This command is used to exit from a recursive editing level back to the top level of EMACS, and to exit from EMACS at top level back to HACTRN. If your keyboard does not have a Meta key, you must type this command by means of a bit prefix character, as C-C C-C or as Altmode C-C. Note carefully the difference between exiting a recursive editing level and aborting it: exiting allows the command which invoked the recursive editing level to finish its job with the text as you have edited it, whereas aborting cancels whatever the command was going to do. See section 24.1 [Aborting], page 129.

We cannot say in general how to exit a subsystem, since each subsystem defines its own command language, but the convention is to use the character "Q".

You can exit from EMACS back to the superior job, usually HACTRN, at any time, even within a recursive editing level, with the command C-X C-C (^R Return to Superior). If this is used while you are inside a recursive editing level, then when EMACS is re-entered you will still be inside the recursive editing level.

Exiting EMACS does not normally save the visited file, because it is not the case that users exit EMACS only when they are "finished editing". If you want the file saved, you must use C-X C-S. Exiting does cause an auto save if auto save mode is in use.

Exiting from EMACS runs the function & Exit EMACS, which executes the value of the variable Exit Hook, if it is defined.

7. Self-Documentation Commands

EMACS provides extensive self-documentation features which revolve around a single character, called the Help character. At any time while using EMACS, you can type the Help character to ask for help. How to type the Help character depends on the terminal you are using, but aside from that the same character always does the trick. If your keyboard has a key labeled Help (above the H), type that key (together with the Top key). Otherwise the way you type the Help character is actually C-_ (Control-Underscore) followed by an H (this is two characters to type, but let's not worry about that). Whatever it is you have to type, to EMACS it is just the Help character. On some terminals just figuring out how to type a Control-Underscore is hard! Typing Underscore and adding the Control key is what is supposed to work, but on some terminals it does not. Sometimes Control-Shift-O works. On VT-100 terminals, Control-/ and Control-? send a Control-_ character.

If you type Help while you are using a subsystem such as INFO, it will give you a list of the commands of that subsystem.

If you type Help in the middle of a multi-character command, it will often tell you about what sort of thing you should type next. For example, if you type M-X and then Help, it will tell you about M-X and how to type the name of the command. If you finish the function name and the Altmode and then type Help, it will tell you about the function you have specified so you can know what arguments it needs. If you type C-X and then type Help, it will tell you about the C-X commands.

But normally, when it's time for you to start typing a new command, Help offers you several options for asking about what commands there are and what they do. It prompts with "Doc (? for help):" at the bottom of the screen, and you should type a character to say what kind of help you want. You could type Help or "?" at this point to find out what options are available. The ones you are most likely to need are described here.

The most basic Help options are Help C and Help D. You can use them to ask what a particular command does. Help C is for character commands; type the command you want to know about after the Help and the "C" ("C" stands for Character). Thus, Help C M-F or Help C Altmode F tells you about the M-F command. Help D is for asking about functions (extended commands); type the name of the function and a Return. Thus, Help D Lisp Mode<cr> tells you about M-X Lisp Mode. "D" stands for "Describe", since Help D actually uses the function Describe to do the work.

A more complicated sort of question to ask is, "what are the commands for working with files"? For this, you can type Help A, followed by the string "file" and a Return. It prints a list of all the functions that have "file" anywhere in their names, including Save All Files, ^R Save File, Append to File, etc. If some of the functions are connected to commands, it will tell you. For example, it would say that you can invoke

^R Save File by typing C-X C-S. "A" stands for "Apropos", since Help A actually uses the function Apropos to do the substring matching. Help A does not list internal functions, only those the nonprogrammer is likely to use. If you want subroutines to be listed as well, you must call Apropos yourself.

Because Apropos looks only for functions whose names contain the string which you specify, you must use ingenuity in choosing substrings. If you are looking for commands for killing backwards and Help A Kill Backwards doesn't reveal any, don't give up. Try just Kill, or just Backwards, or just Back. Be persistent. Pretend you are playing Adventure.

If you aren't sure what characters you accidentally typed to produce surprising results, you can use Help L to find out ("L" stands for "What I ossage"). If you see commands that you don't know, you can use Help C to find out what they did.

If a command doesn't do what you thought you knew it should do, you can ask to see whether it has changed recently. Help N prints out the file called EMACS;EMACS NEWS which is an archive of announcements of changes to EMACS.

To find out about the other Help options, type Help Help. That is, when the first Help asks for an option, type Help to ask what is available.

Finally, you should know about the documentation files for EMACS, which are EMACS;EMACS GUIDE and EMACS;EMACS CHART. EMACS GUIDE is a version of the manual formatted to be printed out on a terminal or line printer. EMACS CHART has a brief description of all the commands, and is good to post on the wall near your terminal.

8. The Mark and the Region

In general, a command which processes an arbitrary part of the buffer must know where to start and where to stop. In EMACS, such commands start at point and end at a place called the "mark". This range of text is called "the region". Here are some commands for setting the mark:

C-@	Set the mark where point is.
C-Space	The same.
C-X C-X	Interchange mark and point.
M-@	Set mark after end of next word.
C-M-@	Set mark after end of next Lisp s-expression.
C-<	Set mark at beginning of buffer.
C->	Set mark at end of buffer.
M-H	Put region around current paragraph.
C-M-H	Put region around current Lisp defun.
C-X H	Put region around entire buffer.
C-X C-P	Put region around current page.

For example, if you wish to convert part of the buffer to all upper-case, you can use the C-X C-U command, which operates on the text in the region. You can first go to the beginning of the text to be capitalized, put the mark there, move to the end, and then type C-X C-U. Or, you can set the mark at the end of the text, move to the beginning, and then type C-X C-U. C-X C-U runs the function ^R Uppercase Region, whose name signifies that the region, or everything between point and the mark, is to be capitalized.

The most common way to set the mark is with the C-@ command or the C-Space command (^R Set/Pop Mark). They set the mark where point is. Then you can move point away, leaving the mark behind.

It isn't actually possible to type C-Space on non-Meta keyboards. Yet on many terminals the command appears to work anyway! This is because trying to type a Control-Space on those terminals actually sends the character C-@, which means the same thing as C-Space. A few keyboards just send a Space. If you have one of them, you suffer, or customize your EMACS.

Since terminals have only one cursor, there is no way for EMACS to show you where the mark is located. You have to remember. The usual solution to this problem is to set the mark and then use it soon, before you forget where it is. But you can see where the mark is with the command C-X C-X (^R Exchange Point and Mark) which puts the mark where point was and point where the mark was. Thus, the previous location of the mark is shown, but the region specified is not changed. C-X C-X is also useful when you are satisfied with the location of point but want to move the other end of the region: do C-X C-X to put point at that end and then you can adjust it. The end of the region which is at point can be moved, while the end which is at the mark stays fixed.

If you insert or delete before the mark, the mark does not stay with the characters it was between. If the buffer contains "FOO BAR" and the mark is before the "B", then if you delete the "F" the mark will be before the "A". This is an unfortunate result of the simple way the mark is implemented. It is best not to delete or insert at places above the mark until you are finished using it and don't care where it drifts to.

8.1. Commands to Mark Textual Objects

There are commands for placing the mark on the other side of a certain object such as a word or a list, without having to move there first. M-@ (^R Mark Word) puts the mark at the end of the next word, while C-M-@ (^R Mark Sexp) puts it at the end of the next s-expression. C-> (^R Mark End) puts the mark at the end of the buffer, while C-< (^R Mark Beginning) puts it at the beginning. These characters allow you to save a little typing, sometimes.

Other commands set both point and mark, to delimit an object in the buffer. M-H (^R Mark Paragraph) puts point at the beginning of the paragraph it was inside of (or before), and puts the mark at the end. M-H does all that's necessary if you wish to indent, case-convert, or kill a whole paragraph. C-M-H (^R Mark Defun) similarly puts point before and the mark after the current or next defun. C-X C-P (^R Mark Page) puts point before the current page (or the next or previous, according to the argument), and mark at the end. The mark goes after the terminating page delimiter (to include it), while point goes after the preceding page delimiter (to exclude it). Finally, C-X H (^R Mark Whole Buffer) makes the region the entire buffer by putting point at the beginning and the mark at the end.

8.2. The Ring of Marks

Aside from delimiting the region, the mark is also useful for remembering a spot that you may want to go back to. To make this feature more useful, EMACS remembers 16 previous locations of the mark. Most commands that set the mark push the old mark onto this stack. To return to a marked location, use C-U C-@ (or C-U C-Space). This moves point to where the mark was, and restores the mark from the stack of former marks. So repeated use of this command moves point to all of the old marks on the stack, one by one. Since the stack is actually a ring, enough uses of C-U C-@ bring point back to where it was originally. Insertion and deletion can cause the saved marks to drift, but they are still good for this purpose because they are approximately right.

Some commands whose primary purpose is to move point a great distance take advantage of the stack of marks to give you a way to undo the command. The best example is M-<, which moves to the beginning of the buffer. It sets the mark first, so that you can use C-U C-@ or C-X C-X to go back to where you were. Searches sometimes set the mark; it depends on how far they move. Because of this uncertainty, searches

type out "^@" if they set the mark. The normal situation is that searches leave the mark behind if they move at least 500 characters, but you can change that value since it is kept in the variable Auto Push Point Option. By setting it to 0, you can make all searches set the mark. By setting it to a very large number such as ten million, you can prevent all searches from setting the mark. The string to be typed out when this option does its thing is kept in the variable Auto Push Point Notification.

9. Killing and Moving Text

The commonest way of moving or copying text with EMACS is to kill it, and get it back again in one or more places. This is very safe because the last several pieces of killed text are all remembered, and it is versatile, because the many commands for killing syntactic units can also be used for moving those units. There are also other ways of moving text for special purposes.

9.1. Deletion and Killing

Most commands which erase text from the buffer save it so that you can get it back if you change your mind, or move or copy it to other parts of the buffer. These commands are known as "kill" commands. The rest of the commands that erase text do not save it; they are known as "delete" commands. The delete commands include C-D and Rubout, which act on single characters, and those commands that delete only spaces or line separators. Commands that can destroy significant amounts of nontrivial data generally kill. The commands' names and individual descriptions use the words "kill" and "delete" to say which they do. If you do a kill command by mistake, you can use the Undo command to undo it (See section 24.3 [Undo], page 132.).

C-D	Delete next character.
Rubout	Delete previous character.
M-\	Delete spaces and tabs around point.
C-X C-O	Delete blank lines around the current line.
M-^	Join two lines by deleting the CRLF and any indentation.
C-K	Kill rest of line or one or more lines.
C-W	Kill region (from point to the mark).
M-D	Kill a word.
M-Rubout	Kill a word backwards.
C-X Rubout	Kill back to beginning of sentence.
M-K	Kill to end of sentence.
C-M-K	Kill s-expression.
C-M-Rubout	Kill s-expression backwards.

9.1.1. Deletion

The most basic delete commands are C-D and Rubout. C-D deletes the character after the cursor - the one the cursor is "on top of" or "underneath". The cursor doesn't move. Rubout deletes the character before the cursor, and moves the cursor back. Line separators act like single characters when deleted. Actually, C-D and Rubout aren't always delete commands; if you give an argument, they kill instead. This prevents you from losing a great deal of text by typing a large argument to a C-D or Rubout.

The other delete commands are those which delete only formatting characters: spaces, tabs and line separators. M-\ (^R Delete Horizontal Space) deletes all the spaces and tab characters before and after point. C-X C-O (^R Delete Blank Lines) deletes all blank lines after the current line, and if the current line is blank deletes all blank lines preceding the current line as well (leaving one blank line, the current line). M-^ (^R Delete Indentation) joins the current line and the previous line, or the current line and the next line if given an argument. See section 11.3 [Indentation], page 46.

A function ^R Delete Region used to exist, but it was too dangerous. When you want to delete a large amount of text without saving a copy of it (perhaps because it is very big), you can set point and mark around the text and then type M-♦ M R K ♦ ♦. (This is a use of the minibuffer. See section 2.3 [Minibuffer], page 127.).

9.1.2. Killing by Lines

The simplest kill command is the C-K command (^R Kill Line). If given at the beginning of a line, it kills all the text on the line, leaving it blank. If given on a blank line, the blank line disappears. As a consequence, if you go to the front of a non-blank line and type two C-K's, the line disappears completely.

More generally, C-K kills from point up to the end of the line, unless it is at the end of a line. In that case it kills the line separator following the line, thus merging the next line into the current one. Invisible spaces and tabs at the end of the line are ignored when deciding which case applies, so if point appears to be at the end of the line, you can be sure the line separator will be killed.

C-K with an argument of zero kills all the text before point on the current line.

If C-K is given a positive argument, it kills that many lines, and the separators that follow them (however, text on the current line before point is spared). With a negative argument, -5 for example, all text before point on the current line, and all of the five preceding lines, are killed.

9.1.3. Other Kill Commands

A kill command which is very general is C-W (^R Kill Region), which kills everything between point and the mark. With this command, you can kill any contiguous characters, if you set the mark at one end of them and go to the other end, first.

Other syntactic units can be killed: words, with M-Rubout and M-D (See section 11.1 [Words], page 43.); s-expressions, with C-M-Rubout and C-M-K (See section 20.6.1 [S-expressions], page 94.); sentences, with C-X Rubout and M-K (See section 11.2 [Sentences], page 44.).

9.2. Un-Killing

Un-killing is getting back text which was killed. The usual way to move or copy text is to kill it and then un-kill it one or more times.

C-Y	Yank (re-insert) last killed text.
M-Y	Replace re-inserted killed text with the previously killed text.
M-W	Save region as last killed text without killing.
C-M-W	Append next kill to last batch of killed text.

Killed text is pushed onto a ring buffer that remembers the last 8 blocks of text that were killed. (Why it is called a "ring buffer" will be explained below). The command C-Y (^R Un-kill) reinserts the text of the most recent kill. It leaves the cursor at the end of the text, and puts the mark at the beginning. Thus, a single C-W undoes the C-Y (M-X Undo also does so). C-U C-Y leaves the cursor in front of the text, and the mark after. This is only if the argument is specified with just a C-U, precisely. Any other sort of argument, including C-U and digits, has an effect described below.

If you wish to copy a block of text, you might want to use M-W (^R Copy Region), which copies the region into the kill ring without removing it from the buffer. This is approximately equivalent to C-W followed by C-Y, except that M-W does not mark the buffer as "changed" and does not temporarily change the screen. Note that there is only one kill ring, and switching buffers or files has no effect on it. After visiting a new file, whatever was last killed in the previous file is still on top of the kill ring.

9.2.1. Appending Kills

Normally, each kill command pushes a new block onto the kill ring. However, two or more kill commands in a row combine their text into a single entry on the ring, so that a single C-Y command gets it all back as it was before it was killed. (Thus we join television in leading people to kill thoughtlessly). If a kill command is separated from the last kill command by other commands, it starts a new entry on the kill ring, unless you tell it not to by saying C-M-W (^R Append Next Kill) in front of it. The C-M-W tells the following command, if it is a kill command, to append the text it kills to the last killed text, instead of pushing a new entry. With C-M-W, you can kill several separated pieces of text and accumulate them to be yanked back in one place.

9.2.2. Un-killing Earlier Kills

To recover text that was killed some time ago (that is, not the most recent victim), you need the Meta-Y (^R Un-kill Pop) command. The M-Y command should be used only after a C-Y command or another M-Y. It takes the un-killed text and replaces it with the text from an earlier kill.

You can think of all the last few kills as living in a ring. After a C-Y command, the text at the front of the ring is also present in the buffer. M-Y "rotates" the ring, bringing the previous string of text to the front, and this text replaces the other text in the buffer as well. Enough M-Y commands can rotate any part of the ring to the front, so you can get at any killed text as long as it is recent enough to be still in the ring. Eventually the ring rotates all the way around and the most recent killed text comes to the front (and into the buffer) again. M-Y with a negative argument rotates the ring backwards. If the region doesn't match the text at the front of the ring, M-Y is not allowed (its definition doesn't make sense in that case).

In any case, when the text you are looking for is brought into the buffer, you can stop doing M-Y's and it will stay there. It's really just a copy of what's at the front of the ring, so editing it does not change what's in the ring. And the ring, once rotated, stays rotated, so that doing another C-Y gets another copy of what you rotated to the front with M-Y.

If you change your mind about un-killing, a C-W or M-X Undo gets rid of the un-killed text at any point, after any number of M-Y's. C-W pushes the text onto the ring again. M-X Undo does not.

If you know how many M-Y's it would take to find the text you want, then there is an alternative. C-Y with an argument greater than one restores the text the specified number of entries down on the ring. Thus, C-U 2 C-Y is gets the next to the last block of killed text. It differs from C-Y M-Y in that C-U 2 C-Y does not permanently rotate the ring.

A way of viewing the contents of the kill ring is

```
M-X View Q-register↑..K<cr>
```

You must add one to the indices listed by this command, to get the argument to use with C-Y to yank any particular string.

9.3. Other Ways of Copying Text

Usually we copy or move text by killing it and un-killing it, but there are other ways that are useful for copying one block of text in many places, or for copying many scattered blocks of text into one place.

9.3.1. Accumulating Text

You can accumulate blocks of text from scattered locations either into a buffer or into a file if you like.

To append them into a buffer, use the command C-X A<buffername><cr> (^R Append to Buffer), which inserts a copy of the region into the specified buffer at the location of point in that buffer. If there is no buffer with the name you specify, one is created, empty. If you append text into a buffer which is visiting a

file, the copied text goes into the middle of the text of the file.

Point in that buffer is left at the end of the copied text, so successive uses of C-X Λ accumulate the text in the specified buffer in the same order as they were copied. If C-X Λ is given an argument, point in the other buffer is left before the copied text, so successive uses of C-X Λ add text in reverse order.

You can retrieve the accumulated text from that buffer with M-X Insert Buffer♦<buffername><cr>. This inserts a copy of the text in that buffer into the selected buffer. You can also select the other buffer for editing. See section 14 [Buffers], page 67, for background information on buffers.

Strictly speaking, C-X Λ does not always append to the text already in the buffer. But if it is used on a buffer which starts out empty, it does keep appending to the end.

To accumulate text into a file, use the command M-X Λppend to File♦<filename><cr>. It adds the text of the region to the end of the specified file. M-X Prepend to File adds the text to the beginning of the file instead. The file is changed immediately on disk. If you wish to insert the text into a copy of the file in an EMACS buffer, you must append to that buffer instead.

9.3.2. Copying Text Many Times

When you want to insert a copy of the same piece of text frequently, the kill ring becomes impractical, since the text moves down on the ring as you edit, and will be in an unpredictable place on the ring when you need it again. For this case, you can use the commands C-X X (^R Put Q-register) and C-X G (^R Get Q-register) to move the text.

C-X X<q> stores a copy of the text of the region in a place called q-register <q>. <q> can be a letter or a number. This gives 36 places in which you can store a piece of text. With an argument, C-X X deletes the text as well. C-X G<q> inserts in the buffer the text from q-register <q>. Normally it leaves point before the text and places the mark after, but with a numeric argument it puts point after the text and the mark before.

The q-registers are important temporary variables in TECO programming, but you don't have to understand them, only to know that what you save with C-X X Λ is what you will get with C-X G Λ.

Do not to use q-registers M and R in this way, if you are going to use the TECO commands MM and MR.

10. Searching

Like other editors, EMACS has commands for searching for an occurrence of a string. The search command is unusual in that it is "incremental"; it begins to search before you have finished typing the search string. As you type in the search string, EMACS shows you where it would be found. When you have typed enough characters to identify the place you want, you can stop.

C-S	Search forward.
C-R	Search backward.
C-S ♦ C-W	Word search, ignoring whitespace.

The command to search is C-S (^R Incremental Search). C-S reads in characters and positions the cursor at the first occurrence of the characters that you have typed. If you type C-S and then F, the cursor moves right after the first "F". Type an "O", and see the cursor move to after the first "FO". After another "O", the cursor is after the first "FOO" after the place where you started the search. At the same time, the "FOO" has echoed at the bottom of the screen.

If you type a mistaken character, you can rub it out. After the FOO, typing a rubout makes the "O" disappear from the bottom of the screen, leaving only "FO". The cursor moves back to the "FO". Rubbing out the "O" and "F" moves the cursor back to where you started the search.

When you are satisfied with the place you have reached, you can type an Altmode, which stops searching, leaving the cursor where the search brought it. Also, any command not specially meaningful in searches stops the searching and is then executed. Thus, typing C-A would exit the search and then move to the beginning of the line. Altmode is necessary only if the next command you want to type is a printing character, Rubout, Altmode or another search command, since those are the characters that would not exit the search.

Sometimes you search for "FOO" and find it, but not the one you expected to find. There was a second FOO that you forgot about, before the one you were looking for. Then type another C-S and the cursor will find the next FOO. This can be done any number of times. If you overshoot, you can rub out the C-S's. You can also repeat the search after exiting it, if the first thing you type after entering another search (when the argument is still empty) is a C-S.

If your string is not found at all, the echo area says "Failing I-Search". The cursor is after the place where EMACS found as much of your string as it could. Thus, if you search for FOOT, and there is no FOOT, you might see the cursor after the FOO in FOOL. At this point there are several things you can do. If your string was mistyped, you can rub some of it out and correct it. If you like the place you have found, you can type Altmode or some other EMACS command to "accept what the search offered". Or you can type C-G, which throws away the characters that could not be found (the "T" in "FOOT"), leaving those that were found (the

"FOO" in "FOOT"). A second C-G at that point undoes the search entirely.

The C-G "quit" command does special things during searches; just what, depends on the status of the search. If the search has found what you specified and is waiting for input, C-G cancels the entire search. The cursor moves back to where you started the search. If C-G is typed while the search is actually searching for something or updating the display, or after search failed to find some of your input (having searched all the way to the end of the file), then only the characters which have not been found are discarded. Having discarded them, the search is now successful and waiting for more input, so a second C-G will cancel the entire search. Make sure you wait for the first C-G to ding the bell before typing the second one; if typed too soon, the second C-G may be confused with the first and effectively lost.

You can also type C-R at any time to start searching backwards. If a search fails because the place you started was too late in the file, you should do this. Repeated C-R's keep looking for more occurrences backwards. A C-S starts going forwards again. C-R's can be rubbed out just like anything else. If you know that you want to search backwards, you can use C-R instead of C-S to start the search, because C-R is also a command (^R Reverse Incremental Search) to search backward. Note to all customizers: all this command does is call the current definition of ^R Incremental Search with a negative argument.

A non-incremental search is also available. Type Altmode right after the C-S to get it. Do M-X Describe♦^R String Search<cr> for details. Some people who prefer non-incremental searches put that function on Meta-S, and ^R Character Search (do M-X Describe♦ for details) on C-S. It can do one useful thing which incremental search cannot: search for words regardless of where the line breaks.

Word search searches for a sequence of words without regard to how the words are separated. More precisely, you type a string of many words, using single spaces to separate them, and the string can be found even if there are multiple spaces or line separators between the words. Other punctuation such as commas or periods must match exactly. This is useful in conjunction with documents formatted by text justifiers. If you edit while looking at the printed, formatted version, you can't tell where the line breaks are in the source file. With word search, you can search without having to know.

Word search is a special case of non-incremental search and is invoked with C-S Altmode C-W. This is followed by the search string, which must always be terminated with an Altmode. Searching does not start until the final Altmode is typed.

11. Commands for English Text

EMACS enables you to manipulate words, sentences, or paragraphs of his text. In addition, there are commands to fill text, and convert case. For text-justifier input files, there are commands that may help manipulate font change commands and underlining.

Editing files of text in a human language ought to be done using Text mode rather than Fundamental mode. Invoke M-X Text Mode to enter Text mode. See section 20 [Major Mode], page 87. M-X Text Mode causes Tab to run the function ^R Tab to Tab Stop, which allows you to set any tab stops with M-X Edit Tab Stops (See section 11.3 [Indentation], page 46.). Features concerned with comments in programs are turned off except when explicitly invoked. Automatic display of parenthesis matching is turned off, which is what most people want. Finally, the syntax table is changed so that periods are not considered part of a word, while apostrophes, backspaces and underlines are.

11.1. Word Commands

EMACS has commands for moving over or operating on words. By convention, they are all Meta-characters.

M-F	Move Forward over a word.
M-B	Move Backward over a word.
M-D	Kill up to the end of a word.
M-Rubout	Kill back to the beginning of a word.
M-@	Mark the end of the next word.
M-T	Transpose two words; drag a word forward or backward across other words.

Notice how these commands form a group that parallels the character based commands C-F, C-B, C-D, C-T and Rubout. M-@ is related to C-@.

The commands Meta-F (^R Forward Word) and Meta-B (^R Backward Word) move forward and backward over words. They are thus analogous to Control-F and Control-B, which move over single characters. Like their Control- analogues, Meta-F and Meta-B move several words if given an argument, and can be made to go in the opposite direction with a negative argument. Forward motion stops right after the last letter of the word, while backward motion stops right before the first letter.

It is easy to kill a word at a time. Meta-D (^R Forward Kill Word) kills the word after point. To be precise, it kills everything from point to the place Meta-F would move to. Thus, if point is in the middle of a word, only the part after point is killed. If some punctuation comes after point and before the next word, it is killed along with the word. If you wish to kill only the next word but not the punctuation, simply do Meta-F

to get the end, and kill the word backwards with Meta-Rubout. Meta-D takes arguments just like Meta-F.

Meta-Rubout (^R Backward Kill Word) kills the word before point. It kills everything from point back to where Meta-B would move to. If point is after the space in "FOO, BAR", "FOO, " is killed. In such a situation, to avoid killing the comma and space, do a Meta-B and a Meta-D instead of a Meta-Rubout.

Meta-T (^R Transpose Words) moves the cursor forward over a word, dragging the word preceding or containing the cursor forward as well. A numeric argument serves as a repeat count. A negative argument undoes the effect of a positive argument; it drags the word behind the cursor backward over a word. An argument of zero, instead of doing nothing, transposes the word at point with the word at mark. In any case, the delimiter characters between the words do not move. For example, "FOO, BAR" transposes into "BAR, FOO" rather than "BAR FOO,".

To operate on the next n words with an operation which applies between point and mark, you can either set the mark at point and then move over the words, or you can use the command Meta-@ (^R Mark Word) which does not move point, but sets the mark where Meta-F would move to. They can be given arguments just like Meta-F. The case conversion operations have alternative forms that apply to words, since they are particularly useful that way.

Note that if you are in Atom Word mode and in Lisp mode, all the word commands regard an entire Lisp atom as a single word. See section 22.1 [Minor Modes], page 111.

The word commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to be a word delimiter. See section 22.4 [Syntax], page 115.

11.2. Sentence and Paragraph Commands

The EMACS commands for manipulating sentences and paragraphs are all Meta- commands, so as to resemble the word-handling commands.

M-A Move back to the beginning of the sentence.
M-E Move forward to the end of the sentence.
M-K Kill this or next sentence.
M-[Move back to previous paragraph beginning.
M-] Move forward to next paragraph end.
M-H Put point and mark around this paragraph (around the following one, if between paragraphs).
C-X Rubout
Kill back to beginning of sentence.

11.2.1. Sentences

The commands Meta-A and Meta-E (^R Backward Sentence and ^R Forward Sentence) move to the beginning and end of the current sentence, respectively. They were chosen to resemble Control-A and Control-E, which move to the beginning and end of a line, but unlike those Control characters Meta-A and Meta-E if repeated move over several sentences. EMACS considers a sentence to end wherever there is a ".", "?" or "!" followed by the end of a line or two spaces, with any number of ")"'s, "]"'s, "'"'s, or '"'s allowed in between. Neither M-A nor M-E moves past the CRLF or spaces which delimit the sentence.

Just as C-A and C-E have a kill command, C-K, to go with them, so M-A and M-E have a corresponding kill commands M-K (^R Kill Sentence) which kills from point to the end of the sentence. With minus one as an argument it kills back to the beginning of the sentence. Larger arguments serve as a repeat count.

There is a special command, C-X Rubout (^R Backward Kill Sentence) for killing back to the beginning of a sentence, because this is useful when you change your mind in the middle of composing text.

11.2.2. Paragraphs

There are similar commands for moving over paragraphs. Meta-[(^R Backward Paragraph) moves to the beginning of the current or previous paragraph, while Meta-] (^R Forward Paragraph) moves to the end of the current or next paragraph. Blank lines and text justifier command lines separate paragraphs and are not part of any paragraph. Also, an indented line starts a new paragraph.

In major modes for programs (as opposed to Text mode), paragraphs are determined only by blank lines. This makes the paragraph commands continue to be useful even though there are no paragraphs per se.

When there is a fill prefix, then paragraphs are delimited by all lines which don't start with the fill prefix.

When you wish to operate on a paragraph, you can use the command Meta-H (^R Mark Paragraph) to prepare. This command puts point at the beginning and mark at the end of the paragraph point was in. Before setting the new mark at the end, a mark is set at the old location of point; this allows you to undo a mistaken Meta-H with two C-U C-@'s. If point is between paragraphs (in a run of blank lines, or at a boundary), the paragraph following point is surrounded by point and mark. Thus, for example, Meta-H C-W kills the paragraph around or after point.

One way to make an "invisible" paragraph boundary that does not show if the file is printed is to put space-backspace at the front of a line. The space makes the line appear (to the EMACS paragraph commands) to be indented, which usually means that it starts a paragraph.

The variable Paragraph Delimiter should be a TECO search string (See section 19.3 [TECO search strings], page 85.) composed of various characters separated by ↑O's. A line whose beginning matches the search string is either the beginning of a paragraph or a text justifier command line part of no paragraph. If the line begins with period, singlequote, "-", "\" or "@", it can be a text justifier command line; otherwise, it can be the beginning of a paragraph; but it cannot be either one unless Paragraph Delimiter is set up to recognize it. Thus, "↑O " as the Paragraph Delimiter string means that lines starting with spaces start paragraphs, lines starting with periods are text justifier commands, and all other nonblank lines are nothing special.

11.3. Indentation Commands for Text

Tab	Indents "appropriately" in a mode-dependent fashion.
M-Tab	Inserts a tab character.
Linefeed	Is the same as Return and Tab.
M-^	Undoes a Linefeed. Merges two lines.
M-M	Moves to the line's first nonblank character.
M-I	Indent to tab stop. In Text mode, Tab does this also.
C-M-\	Indent several lines to same column.
C-X Tab	Shift block of lines rigidly right or left.

The way to request indentation is with the Tab command. Its precise effect depends on the major mode. In Text mode, it indents to the next tab stop. You can set the tab stops with Edit Tab Stops (see below). If you just want to insert a tab character in the buffer, you can use M-Tab.

For English text, usually only the first line of a paragraph should be indented. So, in Text mode, new lines created by Auto Fill mode are not indented. This is brought about by setting the variable Space Indent Flag to zero. This way, Auto Fill can avoid indenting without denying you the use of Tab to indent. But sometimes you want to have an indented paragraph. In such cases, use M-X Edit Indented Text, which enters a submode in which Tab and Auto Fill indent each line under the previous line, and only blank lines delimit paragraphs. Alternatively, you can specify a fill prefix (see below).

To undo a line-break, whether done manually or by Auto Fill, use the Meta-^ (^R Delete Indentation) command to delete the indentation at the front of the current line, and the line boundary as well. They are replaced by a single space, or by no space if before a ")" or after a "(", or at the beginning of a line. To delete just the indentation of a line, go to the beginning of the line and use Meta-\ (^R Delete Horizontal Space), which deletes all spaces and tabs around the cursor.

To insert an indented line before the current one, do C-A, C-O, and then Tab. To make an indented line after the current one, use C-E Linefeed.

To move over the indentation on a line, do Meta-M or C-M-M (^R Back to Indentation). These

commands, given anywhere on a line, position the cursor at the first nonblank character on the line.

There are also commands for changing the indentation of several lines at once. Control-Meta-\ (^R Indent Region) gives each line whose first character is between point and mark the "usual" indentation (as determined by Tab). With a numeric argument, it gives each line precisely that much indentation. C-X Tab (^R Indent Rigidly) moves all of the lines in the region right by its argument (left, for negative arguments).

Usually, EMACS uses both tabs and spaces to indent. If you don't want that, you can use M-X Indent Tabs Mode to turn the use of tabs on or off. To convert all tabs in a file to spaces, you can use M-X Untabify, whose argument is the number of positions to assume between tab stops (default is 8). M-X Tabify performs the opposite transformation, replacing spaces with tabs whenever possible, but only if there are at least three of them so as not to obscure ends of sentences. The visual appearance of the text should never be changed by Tabify or Untabify.

11.3.1. Tab Stops

For typing in tables, you can use Text mode's definition of Tab, ^R Tab to Tab Stop, which may be given anywhere in a line, and indents from there to the next tab stop. If you are not in Text mode, this function can be found on M-I anyway.

Set the tab stops using Edit Tab Stops, which displays for you a buffer whose contents define the tab stops. The first line contains a colon or period at each tab stop. Colon indicates an ordinary tab, which fills with whitespace; a period specifies that characters be copied from the corresponding columns of the second line below it. Thus, you can tab to a column automatically inserting dashes or periods, etc. It is your responsibility to put in the second line the text to be copied. The third and fourth lines you see contain column numbers to help you edit. They are only there while you are editing the tab stops; they are not really part of the tab settings. The first two lines reside in the variable Tab Stop Definitions when they are not being edited. If the second line is not needed, Tab Stop Definitions can be just one line, with no CRLFs. This makes it easier to set the variable in a local modes list. See section 22.6 [Locals], page 118.

11.4. Text Filling

Space	in Auto Fill mode, breaks lines when appropriate.
M-Q	Fill paragraph.
M-G	Fill region (G is for Grind, by analogy with Lisp).
M-S	Center a line.
C-X =	Show current cursor position.

EMACS's Auto Fill mode lets you type in text that is filled (broken up into lines that fit in a specified width) as you go. If you alter existing text and thus cause it to cease to be properly filled, EMACS can fill it again if you ask.

Entering Auto Fill mode is done with M-X Auto Fill. From then on, lines are broken automatically at spaces when they get longer than the desired width. New lines are usually indented, but in Text mode they are not. To leave Auto Fill mode, execute M-X Auto Fill again.

When you finish a paragraph, you can type Space with an argument of zero. This doesn't insert any spaces, but it does move the last word of the paragraph to a new line if it doesn't fit in the old line. Return also moves the last word, but it creates another blank line.

If you edit the middle of a paragraph, it may no longer be correctly filled. To re-fill a paragraph, use the command Meta-Q (^R Fill Paragraph). It causes the paragraph that point is inside, or the one after point if point is between paragraphs, to be re-filled. All the line-breaks are removed, and then new ones are inserted where necessary.

If you are not happy with Meta-Q's idea of where paragraphs start and end (the same as Meta-H's. See section 11.2 [Paragraphs], page 44.), you can use Meta-G (^R Fill Region) which re-fills everything between point and mark. Sometimes, it is ok to fill a region of several paragraphs at once. Meta-G recognizes a blank line or an indented line as starting a paragraph and not fill it in with the preceding line. The sequence space-backspace at the front of a line will prevent it from being filled into the preceding line but is invisible when the file is printed. However, the full sophistication of the paragraph commands in recognizing paragraph boundaries is not available. The purpose of M-G is to allow you to override EMACS's usual criteria for paragraph boundaries.

Giving an argument to M-G or M-Q causes the text to be justified instead of filled. This means that extra spaces are inserted between the words so as to make the right margin come out exactly even. I do not recommend doing this. If someone else has uglified some text by justifying it, you can unjustify it (remove the spaces) with M-G or M-Q without an argument.

The command Meta-S (^R Center Line) centers a line within the current line width. With an argument, it centers several lines individually and moves past them.

The maximum line width for filling is in the variable Fill Column. Both M-Q and Auto Fill make sure that no line exceeds this width. The easiest way to set the variable is to use the command C-X F (^R Set Fill Column) which places the margin at the column point is on, or wherever you specify with a numeric argument. The fill column is initially column 70.

To fill a paragraph in which each line starts with a special marker (which might be a few spaces, giving an

indented paragraph), use the Fill Prefix feature. Move the cursor to a spot right after the special marker and give the command C-X . (^R Set Fill Prefix). Then, filling the paragraph will remove the marker from each line beforehand, and put the marker back in on each line afterward. Auto Fill when there is a fill prefix will insert the fill prefix at the front of each new line. Also, any line which does not start with the fill prefix will be considered to start a paragraph. To turn off the fill prefix, do C-X . with the cursor at the front of a line.

The variable Space Indent Flag controls whether Auto Fill mode indents the new lines that it creates. A nonzero value means that indentation should be done.

The command C-X = (What Cursor Position) can be used to find out the column that the cursor is in, and other miscellaneous information about the cursor which is quick to compute. It prints a line in the echo area that looks like this:

```
X=5 Y=7 CH=101 .=3874(35% of 11014) H=<3051,4640>
```

In this line, the X value is the column the cursor is in (zero at the left), the Y value is the screen line that the cursor is in (zero at the top), the CH value is the octal value of the character after the cursor (101 is "A"), the point value is the number of characters in the buffer before the cursor, and the values in parentheses are the percentage of the buffer before the cursor and the total size of the buffer.

The H values are the virtual buffer boundaries, indicate which part of the buffer is still visible when narrowing has been done. If you have not done narrowing, the H values are omitted. For more information about the virtual buffer boundaries. See section 17 [Narrowing], page 77.

11.5. Case Conversion Commands

EMACS has commands for converting either a single word or any arbitrary range of text to upper case or to lower case.

M-L	Convert following word to lower case.
M-U	Convert following word to upper case.
M-C	Capitalize the following word.
C-X C-L	Convert region to lower case.
C-X C-U	Convert region to upper case.

The word conversion commands are the most useful. Meta-L (^R Lowercase Word) converts the word after point to lower case, moving past it. Thus, successive Meta-L's convert successive words. Meta-U (^R Uppercase Word) converts to all capitals instead, while Meta-C (^R Uppercase Initial) puts the first letter of the word into upper case and the rest into lower case. All these commands convert several words at once if given an argument. They are especially convenient for converting a large amount of text from all upper case to mixed case, because you can move through the text using M-L, M-U or M-C on each word as appropriate.

When given a negative argument, the word case conversion commands apply to the appropriate number of words before point, but do not move point. This is convenient when you have just typed a word in the wrong case. You can give the case conversion command and continue typing.

If a word case conversion command is given in the middle of a word, it applies only to the part of the word which follows the cursor, treating it as a whole word.

The other case conversion commands are C-X C-U (^R Uppercase Region) and C-X C-L (^R Lowercase Region), which convert everything between point and mark to the specified case. Point and mark do not move. These commands ask for confirmation if the region contains more than Region Query Size characters; they also save the original contents of the region so you can undo them (See section 24.3 [Undo], page 132.).

11.6. Font-Changing

EMACS has commands to insert and move font change commands as understood by the TJ6 and R text justifiers. A font change is assumed to be of the form ↑F<digit> meaning select the specified font, or ↑F* meaning select the previously selected font.

M-#	Change previous word's font, or next word's.
C-X #	Change font of region.

M-# is a command to change the font of a word. Its action is rather complicated to describe, but that is because it tries to be as versatile and convenient as possible in practice.

If you type M-# with an argument, the previous word is put into the font specified by the argument. Point is not changed. This means that, if you want to insert a word and put it in a specific font, you can type the word, then use M-# to change its font, and then go on inserting. The font is changed by putting a ↑F<digit> before the word and a ↑F* after.

If you type M-# with no argument, it takes the last font change (either a ↑F<digit> or ↑F*, whichever is later) and moves it one word forward. What this implies is that you can change the font of several consecutive words incrementally by moving after the first word, issuing M-# with an argument to set that word's font, and then typing M-# to extend the font change past more words. Each M-# advances past one more word.

M-# with a negative argument is the opposite of M-# with no argument; it moves the last font change *back* one word. If you type too many M-#'s, you can undo them this way. If you move one font change past another, one or both are eliminated, so as to do the right thing. As a result, M-Minus M-# will undo a M-# with an argument. Try it!

You can also change the font of a whole region by putting point and the mark around it and issuing C-X

#, with the font number as argument. C-X # with a negative argument removes all font changes inside or adjacent to the region.

11.7. Underlining

EMACS has two commands for manipulating text-justifier underlining command characters. These commands do not produce any sort of overprinting in the text file itself; they insert or move command characters which direct text justifiers to produce underlining. By default, commands for the text justifier R are used.

M-_	Underline previous word or next word.
C-X _	Underline region.

M-_ is somewhat like M-# in that it either creates an underline around the previous word or extends it past the next word. However, where a font change requires that you specify a font number, an underline is just an underline and has no parameter for you to specify. Also, it is assumed that the text justifier's commands for starting and ending underlines are distinguishable, whereas you can't tell from a font change whether it is "starting" something or "ending" something. M-_ differs slightly from M-# as a result.

M-_ with no argument creates an underline around the previous word if there is none. If there is an underline there, it is extended one word forward. Thus, you can insert an underlined word by typing the word and then a M-_. Or you can underline several existing words by moving past the first of them, and typing one M-_ for each word.

M-_ given in the vicinity of an underline-begin moves *it*forward. Thus, it should be thought of as applying to any boundary, where underlining either starts or stops, and moving it forward. If a begin underlining is moved past an end, or vice versa, they both disappear.

Giving M-_ an argument merely tells it to apply to several words at once instead of one. M-_ with a positive argument of n underlines the next n words, either creating an underlined area or extending an existing one. With a negative argument, that many previous words are underlined. Thus, M-_ can do more things with underlines than M-# can do with font changes, because of the facts that you don't need to use the argument to say which font, and you can tell a beginning from an end.

For larger scale operations, you can use C-X _ to place underlines from point to mark, or C-X _ with a negative argument to remove all underlining between point and mark.

By default, ↑B is used to begin an underline and ↑E is used to end one. The variables Underline Begin and Underline End may be created and set to strings to use instead. For a single character you can use the numeric ASCII code for it.

12. Commands for Fixing Typos

In this section we describe the commands that are especially useful for the times when you catch a mistake in your text just after you have made it, or change your mind when writing something on line.

Rubout	Delete last character.
M-Rubout	Kill last word.
C-X Rubout	Kill to beginning of sentence.
C-T	Transposes two characters.
C-X C-T	Transposes two lines.
C-X T	Transposes two arbitrary regions.
M-Minus M-L	Convert last word to lower case.
M-Minus M-U	Convert last word to all upper case.
M-Minus M-C	Convert last word to lower case with capital initial.
M-'	Fix up omitted shift key on digit.

12.1. Killing Your Mistakes

The Rubout command is the most important correction command. When used among printing (self-inserting) characters, it can be thought of as canceling the last character typed.

When your mistake is longer than a couple of characters, it might be more convenient to use M-Rubout or C-X Rubout. M-Rubout kills back to the start of the last word, and C-X Rubout kills back to the start of the last sentence. C-X Rubout is particularly useful when you are thinking of what to write as you type it, in case you change your mind about phrasing. M-Rubout and C-X Rubout save the killed text for C-Y and M-Y to retrieve (See section 9.2 [Un-killing], page 37.).

M-Rubout is often useful even when you have typed only a few characters wrong, if you know you are confused in your typing and aren't sure exactly what you typed. At such a time, you cannot correct with Rubout except by looking at the screen to see what you did. It requires less thought to kill the whole word and start over again, especially if the system is heavily loaded.

12.2. Transposition

The common error of transposing two characters can be fixed, when they are adjacent, with the C-T command. Normally, C-T transposes the two characters on either side of the cursor. When given at the end of a line, rather than transposing the last character of the line with the line separator, which would be useless, C-T transposes the last two characters on the line. So, if you catch your transposition error right away, you can fix it with just a C-T. If you don't catch it so fast, you must move the cursor back to between the two

transposed characters. If you transposed a space with the last character of the word before it, the word motion commands are a good way of getting there. Otherwise, a reverse search (C-R) is often the best way. See section 10 [Search], page 41.

To transpose two lines, use the C-X C-T command (^R Transpose Lines). M-T transposes words and C-M-T transposes s-expressions.

A more general transpose command is C-X T (^R Transpose Regions). This transposes two arbitrary blocks of text, which need not even be next to each other. To use it, set the mark at one end of one block, then at the other end of the block; then go to the other block and set the mark at one end, and put point at the other. In other words, point and the last three marks should be at the four locations which are the ends of the two blocks. It does not matter which of the four locations point is at, or which order the others were marked. C-X T transposes the two blocks of text thus identified, and relocates point and the three marks without changing their order.

12.3. Case Conversion

A very common error is to type words in the wrong case. Because of this, the word case-conversion commands M-L, M-U and M-C have a special feature when used with a negative argument: they do not move the cursor. As soon as you see you have mistyped the last word, you can simply case-convert it and go on typing. See section 11.5 [Case], page 49.

Another common error is to type a special character and miss the shift key, producing a digit instead. There is a special command for fixing this: M-' (^R Upcase Digit), which fixes the last digit before point in this way (but only if that digit appears on the current line or the previous line. Otherwise, to minimize random effects of accidental use, M-' does nothing). Once again, the cursor does not move, so you can use M-' when you notice the error and immediately continue typing. Because M-' needs to know the arrangement of your keyboard, the first time you use it you must supply the information by typing the row of digits 1, 2, ... , 9, 0 but *holding down the shift key*. This tells M-' the correspondence between digits and special characters, which is remembered for the duration of the EMACS. This command is called M-' because its main use is to replace "7" with a single-quote.

13. File Handling

The basic unit of stored data is the file. Each program, each paper, lives usually in its own file. To edit a program or paper, the editor must be told the name of the file that contains it. This is called "visiting" the file. To make your changes to the file permanent on disk, you must "save" the file. EMACS also has facilities for deleting files conveniently, and for listing your file directory. Special text in a file can specify the modes to be used when editing the file.

13.1. Visiting Files

C-X C-V	Visit a file.
C-X C-R	Visit a file for reading only.
C-X C-S	Save the visited file.
Meta-~	Tell EMACS to forget that the buffer has been changed.

Visiting a file means copying its contents into EMACS where you can edit them. EMACS remembers the name of the file you visited. Unless you use the multiple buffer and window features of EMACS, you can only be visiting one file at a time. The name of the file you are visiting in the currently selected buffer is visible in the mode line when you are at top level.

The changes you make with EMACS to the text of the file you are visiting are made not in the file itself, but in a copy inside EMACS. The file itself is not changed. The changed text is not permanent until you save it in a file. The first time you change the text, a star appears at the end of the mode line; this indicates that the text contains fresh changes which will be lost unless you save them. You can do that at any time with C-X C-S. If you change one file and then try to visit another in the same buffer, EMACS offers to save the first file (if it is not saved, the changes are lost). In addition, for those who are afraid of system crashes, Auto Save mode saves the file at regular intervals automatically while you edit. See section 13.3 [Auto Save], page 57. Journal files are another way of protecting against crashes. See section 24.4 [Journals], page 133.

To visit a file, use the command C-X C-V (^R Visit File). Follow the command with the name of the file you wish to visit, terminated by a Return. If you can see a filename in the mode line, then that name is the default, and any component of the filename which you don't specify is taken from them. If EMACS thinks you can't see the defaults, they are included in the prompt. You can abort the command by typing C-G, or edit the filename with Rubout and C-U. If you do type a Return to finish the command, the new file's text appears on the screen, and its name shows up in the mode line.

When you wish to save the file and make your changes permanent, type C-X C-S (^R Save File). After the save is finished, C-X C-S prints "Written: <filenames>" in the echo area at the bottom of the screen. If

you are visiting a file whose name is ">", this message contains the version number actually written. If there are no changes to save, the file is not saved; it would be redundant to save a duplicate of the previous version.

However, you need not do the saves yourself. If you alter one file and then visit another, EMACS may offer to save the old one. If you answer Y, the old file is saved; if you answer N, all the changes you have made to it since the last save are lost. You should not type ahead after a file visiting command, because your type-ahead might answer an unexpected question in a way that you would regret. If you are sure you only want to look at a file, and not change it, you can use the C-X C-R command to visit it, instead of C-X C-V. If a file was visited with C-X C-R, EMACS does not offer to save it when you visit the next file. It assumes the changes were inadvertent. However, you can still save the file with C-X C-S.

If EMACS is about to save a file automatically and discovers that the text is now a lot shorter than it used to be, it tells you so and asks for confirmation (Y or N). If you aren't sure what to answer (because you are surprised that it has shrunk), type C-G to abort everything, and take a look at your buffer.

Sometimes you will change a buffer by accident. Even if you undo the change (perhaps, rub out the character you inserted), EMACS still knows that "the buffer has been changed". You can tell EMACS to forget about that with the Meta-~ (^R Buffer Not Modified) command. This command simply clears the "modified" flag which says that the buffer contains changes which need to be saved. It is up to you not to use it unwisely. If we take " -" to mean "not", then Meta-~ is "not" metafied.

What if you want to create a file? Just visit it. EMACS print "(New File)" but otherwise acts unworried. If you make any changes and save them, the file is created. If you visit a nonexistent file unintentionally (because you typed the wrong file name), visit the file you meant. If you didn't "change" the nonexistent file (you never inserted anything in it), it is not created.

If EMACS is about to save a file and sees that the latest version on disk is not the same one as EMACS last read or wrote, EMACS notifies you of this fact, and asks what to do, because this probably means that something is wrong. For example, someone else may have been editing the same file. If this is so, there is a good chance that your work or his work will be lost if you don't take the proper steps. You should first find out exactly what is going on. The C-X C-D command to list the directory will help. If you determine that someone else has modified the file, save your file under different names (or at least making a new version) and then SRCCOM the two files to merge the two sets of changes. Also get in touch with the other person so that he doesn't continue editing.

13.2. How to Undo Drastic Changes to a File

If you have made extensive changes to a file and then change your mind about them, you can get rid of them by reading in the previous version of the file. To do this, use M-X Revert File. If you have been using Auto Save mode, it reads in the last version of the visited file or the last auto save file, whichever is more recent.

If you are using Auto Save mode, saving as special Auto Save filenames, then you can ask to revert to the last "real" save, ignoring subsequent auto saves, with C-U M-X Revert File. If you are using the style of auto saving which saves under the real filenames, this is not possible.

M-X Revert File does not change point, so that if the file was only edited slightly, you will be at approximately the same piece of text after the Revert as before. If you have made drastic changes, the same value of point in the old file may address a totally different piece of text.

Because M-X Revert File can be a disaster if done by mistake, it asks for confirmation (Y or N) before doing its work. A pre-comma argument can be used to inhibit the request for confirmation when you call the function Revert File from a TECO program, as in 1,M(M.M Revert_File♦).

13.3. Auto Save Mode: Protection Against Crashes

If you turn on Auto Save mode, EMACS saves your file from time to time (based on counting your commands) without being asked. Your file is also saved if you stop typing for more than a few minutes when there are changes in the buffer. This prevents you from losing more than a limited amount of work when the system crashes. (Another method of protection against crashes is the journal file. See section 24.4 [Journals], page 133.). You can turn auto saving on or off in an individual buffer with M-X Auto Save. In addition, you can have auto saving by default in all buffers by setting the option Auto Save Default. The frequency of saving, and the number of saved versions to keep, can both be specified.

Each time you visit a file, no matter how, auto saving will be on for that file if Auto Save Default is nonzero. However, by giving a nonzero argument to the file-visiting command, you can turn off auto saving *for that file only*, without changing the default. For example, you might use C-U C-X C-V to do this. Once you have visited a file, you can turn auto saving on or off with M-X Auto Save. Like other minor mode commands, M-X Auto Save turns the mode on with a positive argument, off with a zero or negative argument; with no argument, it toggles. If you start typing a new file into a buffer without visiting anything, auto save mode is initially off, but you can turn it on with M-X Auto Save.

When an auto save happens, "(Auto Save)" is printed in the echo area (On a printing terminal, the bell is

rung instead). An error in the process of auto saving prints "(Auto Save Error!)".

Let us suppose that it is time for an automatic save to be done: where should the file be saved?

Two workable methods have been developed: save the file under the names you have visited, or save it under some special "auto save file name". Each solution has its good and bad points. The first one is excellent some of the time, but intolerable the rest of the time. The second is usually acceptable. Auto saving under the visited file's actual names means that you need do nothing special to gobble the auto save file when you need it; and it means that there is no need to worry about interference between two users sharing a directory, as long as they aren't editing the same file at once. However, this method can sometimes have problems:

If you visit a file on a device other than DSK, auto saves can't go there, because it would probably be slow.

If your file does not have a numeric version number, or you have visited a fixed version, auto saves can't go under that name, because they would clobber the original file.

If you visit a file with C-X C-R, then you have said you don't want to store under those names.

If you haven't visited a file, there aren't any names to use.

In all these cases, the filenames for auto saving are taken from the variable Auto Save Filenames. If none of those cases apply then it is possible to store auto saves under the visited name. This will be done, provided that you turn on the feature by setting the variable Auto Save Visited File to a nonzero value.

When you want to save your file "for real", use C-X C-S, as always. C-U C-X C-S is a way to request an "auto" save explicitly. When you are auto saving under the visited filenames, there is not much difference between an auto save and a "real" save, except that an auto save will eventually be deleted automatically by EMACS a few auto saves later, while a "real" save will be left around forever (at least, auto save won't delete it).

When it is time to recover from a system crash by reloading the auto save file, if auto saving was using the visited file names you have nothing special to do. If auto saving was using special auto save filenames, read in the last auto save file and then use C-X C-W (Write File) to write it out in its real location. If you want to use an auto save file to throw away changes that you don't like, you can use M-X Revert File, which knows how to find the most recent save, permanent or not, under whatever filenames. See section 13.2 [Revert], page 57.

For your protection, if a file has shrunk by more than 30% since the last save, auto saving does not save. Instead it prints a message that the file has shrunk. You can save explicitly if you wish; after that, auto saving will resume.

Although auto saving generates large numbers of files, it does not clog directories, because it cleans up after itself. Only the last Auto Save Max auto save files are kept; as further saves are done, old auto saves are

deleted. However, files which were not made by auto saving (or by explicitly requested auto-saves with C-U C-X C-S) are never deleted in this way. The variable Auto Save Max is initially 2. Changing the value may not take effect in a given buffer until you turn auto saving off and on in that buffer.

The number of characters of input between auto saves is controlled by the variable Auto Save Interval. It is initially 500. Changing this takes effect immediately.

Auto Save Filenames is usually set up by the default init file to DSK:<working directory>;_^RSV >. If you use auto saving in multiple buffers a lot, you might want to have a Buffer Creation Hook which sets Auto Save Filenames to a filename based on the buffer name, so that different buffers don't interfere with each other.

13.4. Listing a File Directory

To look at a part of a file directory, use the C-X C-D command (^R Directory Display). With no argument, it shows you the file you are visiting, and related files with the same first name. C-U C-X C-D reads a filename from the terminal and shows you the files related to that filename.

To see the whole directory in a brief format, use the function List Files, which takes the directory name as a string argument. The function View Directory prints a verbose listing of a whole directory.

The variable Auto Directory Display can be set to make many file operations display the directory automatically. The variable is normally 0; making it positive causes write operations such as Write File to display the directory, and making it negative causes read operations such as Insert File or visiting to display it as well. The display is done using the default directory listing function which is kept in the variable Directory Lister. Normally, in EMACS, this is the function that displays only the files related to the current default file. An alternative type of directory listing can be obtained by setting Directory Lister to M-M-& Rotated Directory Listing♦. This function always displays the whole directory, but starts with the file you are interested in, proceeding through the end of the directory around to the beginning.

13.5. Cleaning a File Directory

The normal course of editing constantly creates new versions of files. If you don't eventually delete the old versions, the directory will fill up and further editing will be impossible. EMACS has commands that make it easy to delete the old versions.

For complete flexibility to delete precisely the files you want to delete, you can use the DIRED package. See section 13.6 [DIRED], page 60, for more details.

But there is a more convenient way to do the usual thing: keep only the two (or other number) most recent versions.

M-X Reap File♦<file><cr> counts the number of versions of <file>. If there are more than two, you are told the names of the recent ones (to be kept) and the names of the older ones (to be deleted), and asked whether to do the deletion (answer Y or N). Files which have the "Don't Reap" bit set are excluded; they are always kept.

Reap File makes a special offer to delete individual files whose FN2 indicates that they are likely to be temporary. The list of temporary names is contained in a TECO search string in the variable Temp File FN2 List. See section 19.3 [TECO search strings], page 85.

If you give M-X Reap File a null filename argument, or no argument, then it applies to the file you are visiting.

M-X Clean Directory♦<dirname>; <cr> cleans a whole directory of old versions. Each file in the directory is processed a la M-X Reap File. M-X Clean Dir with a null argument, or no argument, cleans the directory containing the file you are visiting.

M-X Reap File and M-X Clean Dir can be given a numeric argument which specifies how many versions to keep. For example, C-U 4 M-X Reap File would keep the four most recent versions. The default when there is no argument is the value of the variable File Versions Kept, which is initially 2.

13.6. DIRED, the Directory Editor Subsystem

DIRED makes it easy to delete many of the files in a single directory at once. It presents a copy of a listing of the directory, which you can move around in, marking files for deletion. When you are satisfied, you can tell DIRED to go ahead and delete the marked files.

Invoke DIRED with M-X DIRED to edit the current default directory, or M-X DIRED♦<dir>;<cr> to edit directory <dir>. You are then given a listing of the directory which you can move around in with all the normal EMACS motion commands. Some EMACS commands are made illegal and others do special things, but it's still a recursive editing level which you can exit normally with C-M-C and abort with C-].

You can mark a file for deletion by moving to the line describing the file and typing D, C-D, K, or C-K. The deletion mark is visible as a D at the beginning of the line. Point is moved to the beginning of the next line, so that several D's delete several files. Alternatively, if you give D an argument it marks that many consecutive files. Given a negative argument, it marks the preceding file (or several files) and puts point at the first (in the buffer) line marked. Most of the DIRED commands (D, U, !, $, P, S, C, E, Space) repeat this

way with numeric arguments.

If you wish to remove a deletion mark, use the U (for Undelete) command, which is invoked like D: it removes the deletion mark from the current line (or next few lines, if given an argument). The Rubout command removes the deletion mark from the previous line, moving up to that line.. Thus, a Rubout after a D precisely cancels the D.

For extra convenience, Space is made a command similar to C-N. Moving down a line is done so often in DIRED that it deserves to be easy to type. Rubout is often useful simply for moving up.

If you are not sure whether you want to delete a file, you can examine it by typing E. This enters a recursive editing mode on the file, which you can exit with C-M-C. The file is not really visited at that time, and you are not allowed to change it. When you exit the recursive editing level, you return to DIRED. The V command is like E but uses View File to look at the file.

When you have marked the files you wish to mark, you can exit DIRED with C-M-C. If any files were marked for deletion, DIRED lists them in a concise format, several per line. A file with "!" appearing next to it in this list has not been saved on tape and will be gone forever if deleted. A file with ">" in front of it is the most recent version of a sequence and you should be wary of deleting it. Then DIRED asks for confirmation of the list. You can type "YES" (just "Y" won't do) to go ahead and delete them, "N" to return to editing the directory so you can change the marks, or "X" to give up and delete nothing. No Return character is needed. Anything else typed makes DIRED print a list of these responses and try again to read one of them.

13.6.1. Other DIRED Commands

The "!" command moves down (or up, with an argument of -1) to the next undumped file (one with a "!" before its date).

N finds the next "hog": the next file which has at least three versions.

T when given on a line describing a link marks for deletion the file which the link points to. This file need not be in the directory you are editing to be deleted in this way.

S copies the file you are pointing at to the secondary pack.

P copies the file you are pointing at to the primary pack.

$ complements the don't-reap attribute of the file; this is displayed as a dollar sign to the left of the file date.

M moves the file to another directory or device. You must say where to move it.

C runs SRCCOM to compare the file version you are pointing at with the latest version of the same file. You must confirm the SRCCOM command with a Return before it is executed; you can add extra SRCCOM switches before the Return. When you return to EMACS, the cursor moves down a line to the next file.

H helps you clean up. It marks "old" versions of the current file, and versions with "temporary" second file names, for deletion. You can then use the D and U commands to add and remove marks before deleting the files. The variables File Versions Kept and Temp File FN2 List control which files H picks for deletion. With an argument (C-U H), it does the whole directory instead of just the current file.

? displays a list of the DIRED commands.

13.6.2. Invoking DIRED

There are some other ways to invoke DIRED. The Emacs command C-X D puts you in DIRED on the directory containing the file you are currently editing. With a numeric argument of 1 (C-U 1 C-X D), only the current file is displayed instead of the whole directory. In combination with the H command this can be useful for cleaning up excess versions of a file after a heavy editing session. With a numeric argument of 4 (C-U C-X D), it asks you "Directory:". Type a directory name followed by a semicolon, and/or a file name. If you explicitly specify a file name only versions of that file are displayed, otherwise the whole directory is displayed.

13.6.3. Editing the DIRED Buffer Yourself

It is unwise to try to edit the text of the directory listing yourself, without using the special DIRED commands, unless you know what you are doing, since you can confuse DIRED that way. To make it less likely that you will do so accidentally, the self-inserting characters are all made illegal inside DIRED. However, deleting whole lines at a time is certainly safe. This does not delete the files described by those lines; instead, it makes DIRED forget that they are there and thus makes sure they will *not* be deleted. Thus, M-X Keep Lines^ is useful if you wish to delete only files with a FOO in their names. See section 19 [Replacement], page 83.

For more complicated things, you can use the minibuffer. When you call the minibuffer from within DIRED, you get a perfectly normal one. The special DIRED commands are not present while you are editing in the minibuffer. To mark a file for deletion, replace the space at the beginning of its line with a "D". To remove a mark, replace the "D" with a space.

13.7. Miscellaneous File Operations

EMACS has extended commands for performing many other operations on files.

M-X View File♦ <file> <cr> allows you to scan or read a file by sequential screenfuls without visiting the file. It enters a subsystem in which you type a Space to see the next screenful or a Backspace to see the previous screenful. Typing anything else exits the command. View File does not visit the file; it does not alter the contents of any buffer. The advantage of View File is that the whole file does not need to be loaded before you can begin reading it. The inability to do anything but page forward or backward is a consequence.

M-X Write File♦ <file> <cr> writes the contents of the buffer into the file <file>, and then visits that file. It can be thought of as a way of "changing the name" of the file you are visiting. C-X C-W is another way of getting at this command.

M-X Insert File♦ <file> <cr> inserts the contents of <file> into the buffer at point, leaving point unchanged before the contents and mark after them. The current defaults are used for <file>, and are updated.

M-X Write Region♦ <file> <cr> writes the region (the text between point and mark) to the specified file. It does not set the visited filename. The buffer is not changed.

M-X Append to File♦ <file> <cr> appends the region to <file>. The text is added to the end of <file>.

M-X Prepend to File♦ <file> <cr> adds the text to the beginning of <file> instead of the end.

M-X Set Visited Filename♦<file><cr> changes the name of the file being visited without reading or writing the data in the buffer. M-X Write File is equivalent to this command followed by a C-X C-S.

M-X List Files♦<dir spec><cr> lists just the names of all the files in <dir>, several to a line.

M-X Delete File♦<file><cr> deletes the file.

M-X Copy File♦<old file>♦<new file><cr> copies the file.

M-X Rename File♦<old name>♦<new name><cr> renames the file.

The default filenames for all of these operations are the "TECO default filenames". Most of these operations also leave the TECO default names set to the file they operated on. The TECO default is *not always* the same as the file you are visiting. When you visit a file, they start out the same; the commands mentioned above change the TECO default, but do not change the visited filenames. Each buffer has its own TECO default filenames.

The operation of visiting a file is available as a function under the name M-X Visit File♦<file>♦. In this

form, it uses the TECO default as its defaults, though it still sets both the TECO default and the visited filenames.

13.8. The Directory Comparison Subsystem

The function Compare Directories makes it easy to compare two directories to see which files are present in both and which are present only in one. It compares a directory on the local machine with the directory of the same name on another machine.

Do M-X Compare Directories↓<machine>:<dir spec> <switch>↓, where <machine> is AI, ML, MC or DM, and is not the machine you are on. <dir spec> is an optional directory name and semicolon, and the optional switch is a slash followed by S, D or L.

After a while of crunchino you will be placed in a recursive editing level on a listing of both directories. The reason for the recursive e ting level is simply to make it easy for you to view the comparison; unlike DIRED, Compare Directories does not have any commands for moving or deleting the files. To exit, do C-M-C.

Here is a sample of part of a directory comparison:

```
AI    RMS     #1=72 #2=78 #3=71 #4=77 #5=87 -
MC    RMS     #0=231 #1=254 #13=2844
AI MC       .DDT.  (INIT)     1    11/18/76 01:08    10/21/76 05:06
AI MC L     .DDT←  (INIT) STAN.K ML EXIT
   MC L     .TECO. (INIT) .TECO. .TECO. (INIT)
AI           AR2    1          16    2/6/77    17:51
AI           AR3    1          13    2/17/77   21:37
AI     L     ATS    ORDER   .INFO. @ ORDER
   MC        FTPU   4          9                       !3/4/77    16:46
   MC        FTPU   5          9                       !3/4/77    16:49
AI MC        MATCH  1          15   !3/4/77    15:39   !3/4/77    15:39
```

It begins with one line for each of the two directories, these lines say which two directories they are, and how much disk space is available on each of the machines.

Then there comes the list of files, one line for each distinct pair of filenames that appears. At the beginning of the line appear the names of the machines on which the file exists. At the end of the line come the creation dates (or names pointed at, for links) of the file for the machines it is on. Note that all the dates/link names for the first machine line up, and so do all those for the second machine.

The switches allow you to view only some of the files. The /S switch shows only files present on both machines. /D shows only those not present on both machines. /L shows only files which are the most recent

(largest-numbered) of a sequence. Only one switch is allowed.

14. Using Multiple Buffers

C-X B	Select or create a buffer.
C-X C-F	Visit a file in a new buffer.
C-X C-B	List the existing buffers.
C-X K	Kill a buffer.

When we speak of "the buffer", which contains the text you are editing, we have given the impression that there is only one. In fact, there may be many of them, each with its own body of text. At any time only one buffer can be "selected" and available for editing, but it isn't hard to switch to a different one. Each buffer individually remembers which file it is visiting, what modes are in effect, and whether there are any changes that need saving.

Each buffer in EMACS has a single name, which normally doesn't change. A buffer's name can be any length. The name of the currently selected buffer, and the name of the file visited in it, are visible in the mode line when you are at top level. A newly started EMACS has only one buffer, named "Main".

As well as the visited file and the major mode, a buffer can, if ordered to, remember many other things "locally", which means, independently of all other buffers. See section 22.3 [Variables], page 114.

14.1. Creating and Selecting Buffers

To create a new buffer, you need only think of a name for it (say, "FOO") and then do C-X B FOO<cr>, which is the command C-X B (Select Buffer) followed by the name. This makes a new, empty buffer and select it for editing. The new buffer is not visiting any file, so if you try to save it you will be asked for the filenames to use. Each buffer has its own major mode; the new buffer's major mode is taken from the value of the variable Default Major Mode, or from the major mode of the previously selected buffer if that variable is the null string. Normally this is Fundamental mode.

To return to buffer FOO later after having switched to another, the same command C-X B FOO<cr> is used, since C-X B can tell whether a buffer named FOO exists already or not. It does not matter whether you use upper case or lower case in typing the name of a buffer. C-X B Main<cr> reselects the buffer Main that EMACS started out with. Just C-X B<cr> reselects the previous buffer. Repeated C-X B<cr>'s alternate between the last two buffers selected.

You can also read a file into its own newly created buffer, all with one command: C-X C-F, followed by the filename. The first name of the file becomes the buffer name. C-F stands for "Find", because if the specified file already resides in a buffer in your EMACS, that buffer will be reselected. So you need not remember whether you have brought the file in already or not. A buffer created by C-X C-F can be

reselected later with C-X B or C-X C-F, whichever you find more convenient. Nonexistent files can be created with C-X C-F just as they can be with C-X C-V.

If the buffer with the same name that C-X C-F wants to use already exists but with the wrong contents (often a different file with a similar name), then you are asked what to do. You can type Return meaning go ahead and use the buffer for this new file, or you can type another buffer name to use instead of the normal one. If C-X C-F does find the file already in a buffer, then it checks to see whether the version on disk is the same as the last version read or written from that buffer, for safety. If they are different, you are warned that someone else may be editing the file, and left with the version which was already in the EMACS. To get the new version from disk instead, use M-X Revert File.

14.2. Using Existing Buffers

To get a list of all the buffers that exist, do C-X C-B (List Buffers). Each buffer's name, major mode, and visited filenames are printed. A star at the beginning of a line indicates a buffer which contains changes that have not been saved. The number that appears before a buffer's name in a C-X C-B listing is that buffer's "buffer number". You can select a buffer by giving its number as a numeric argument to C-X B, which then does not need to read a string from the terminal.

If several buffers have stars, you should save some of them with M-X Save All Files. This finds all the buffers that need saving and asks about each one individually. Saving the buffers this way is much easier and more efficient than selecting each one and typing C-X C-S.

A quick way of glancing at another buffer, faster than selecting it, is to use M-X View Buffer↵<buffername><cr>. This displays the contents of the other buffer and lets you move forward and back a screen at a time with Space and Backspace. See section 15 [Display], page 71.

M-X Rename Buffer↵<new name><cr> changes the name of the currently selected buffer. If <new name> is the null string, the first filename of the visited file is the used for the new name of the buffer.

The commands C-X A (^R Append to Buffer) and M-X Insert Buffer can be used to copy text from one buffer to another. See section 9.3 [Copying], page 38.

14.3. Killing Buffers

After you use an EMACS for a while, it may fill up with buffers which you no longer need. Eventually you can reach a point where trying to create any more results in an "URK" error. So whenever it is convenient you should do M-X Kill Some Buffers, which asks about each buffer individually. You can say Y

or N to kill it or not. Or you can say Control-R to take a look at it first. This does not actually select the buffer, as the mode line shows, but gives you a recursive editing level in which you can move around and look at things. When you have seen enough to make up your mind, exit the recursive editing level with a C-M-C and you will be asked the question again. If you say to kill a buffer that needs saving, you will be asked whether it should be saved.

You can kill the buffer FOO by doing C-X K FOO<cr>. You can kill the current buffer, a common thing to do if you use C-X C-F, by doing C-X K<cr>. If you kill the current buffer, in any way, EMACS asks you which buffer to select instead. Saying just <cr> at that point tells EMACS to choose one reasonably. C-X K runs the function Kill Buffer.

15. Controlling the Display

C-L	Clear and redisplay screen, with point at specified place.
C-V	Scroll forwards (a screen or a few lines).
M-V	Scroll backwards.
M-R	Move point to given vertical position.
C-M-R	Get this function onto the screen.

The terminal screen is rarely large enough to display all of your file. If the whole buffer doesn't fit on the screen, EMACS shows a contiguous portion of it, containing point. It continues to show approximately the same portion until point moves outside of it; then EMACS chooses a new portion centered around the new point. This is EMACS's guess as to what you are most interested in seeing. But if the guess is wrong, you can use the display control commands to see a different portion. The finite area of screen through which you can see part of the buffer is called "the window", and the choice of where in the buffer to start displaying is also called "the window".

The basic display control command is C-L (^R New Window). In its simplest form, with no argument, it clears the screen and tells EMACS to display a portion of the buffer centered around where point is currently located (actually, point is placed 35% of the way down from the top; this percentage is controlled by the flag FS % CENTER♦, whose value is the percent of the screen down from the top. See section 22.5 [FS Flags], page 117.).

C-L with a positive argument chooses a new window so as to put point that many lines from the top. An argument of zero puts point on the very top line. Point does not move with respect to the text; rather, the text and point move rigidly on the screen. C-L with a negative argument puts point that many lines from the bottom of the window. For example, C-U -1 C-L puts point on the bottom line, and C-U -5 C-L puts it five lines from the bottom. C-L with an argument does not clear the screen, so that it can move the text on the screen without sending it again if the terminal allows that.

C-U C-L is different from C-L with any other sort of argument. It causes the line containing point to be redisplayed but not the whole screen.

If you want to see a few more lines at the bottom of the screen and don't want to guess what argument to give to C-L, you can use the C-V (^R Next Screen) command. C-V with an argument shows you that many more lines at the bottom of the screen, moving the text and point up together as C-L might. C-V with a negative argument shows you more lines at the top of the screen, as does Meta-V (^R Previous Screen) with a positive argument.

Often you want to read a long file sequentially. For this, the C-V command without an argument is ideal;

it takes the last two lines at the bottom of the screen and puts them at the top, followed by nearly a whole screenful of lines not visible before. Point is put at the top of the screen. Thus, each C-V shows the "next screenful", except for two lines of overlap to provide continuity. The variable Next Screen Context Lines, if defined, controls how many lines from the bottom of the screen move to the top; the default if the variable is not defined is 2. To move backward, use M-V without an argument, which moves a whole screenful backwards (again with overlap).

Scanning by screenfuls through the buffer for some distance is most conveniently done with the M-X View Buffer command. This command enters a simple subsystem in which Space moves a screenful forward and Backspace moves a screenful backward. The Return character exits, leaving point centered in whatever part of the buffer was visible. Any other character exits and returns point to its former location, and is then executed as a command (unless it is a Rubout; Rubout exits but is not executed). View Buffer can be used to view another buffer by giving the buffer's name as a string argument. In this case, exiting with Return moves point permanently in the other buffer, but does not select it. See section 14 [Buffers], page 67.

You can also scan by screenfuls through a file which you have not visited with the M-X View File command. See section 13.7 [Advanced File Commands], page 63.

To scroll the buffer so that the current function or paragraph is positioned conveniently on the screen, use the C-M-R command (^R Reposition Window). This command tries to get as much as possible of the current function or paragraph onto the screen, preferring the beginning to the end, but not moving point off the screen. A "function" in Lisp mode is a defun; otherwise it is defined to be a set of consecutive unindented lines, or a set of consecutive indented lines.

C-L in all its forms changes the position of point on the screen, carrying the text with it. Another command moves point the same way but leaves the text fixed. It is called Meta-R (^R Move to Screen Edge). With no argument, it puts point at the center of the screen. An argument is used to specify the line to put it on, counting from the top if the argument is positive, or from the bottom if it is negative. Thus, Meta-R with an argument of 0 puts the cursor on the top line of the screen. Meta-R never causes any text to move on the screen; it causes point to move with respect to the screen and the text.

16. Two Window Mode

C-X 2	Start showing two windows.
C-X 3	Show two windows but stay "in" the top one.
C-X 1	Show only one window again.
C-X O	Switch to the Other window
C-X 4	Find buffer, file or tag in other window.
C-X ^	Make this window bigger.
C-M-V	Scroll the other window.

Normally, EMACS is in "one-window mode", in which a single body of text is visible on the screen and can be edited. At times, it is useful to have parts of two different files visible at once. For example, while adding to a program a use of an unfamiliar feature, one might wish to see the documentation of that feature at the same time. Two-window mode makes this possible.

The command C-X 2 (^R Two Windows) enters two-window mode. A line of dashes appears across the middle of the screen, dividing the text display area into two halves. Window one, containing the same text as previously occupied the whole screen, fills the top half, while window two fills the bottom half. The cursor moves to window two. If this is your first entry to two-window mode, window two will contain a new buffer named W2. Otherwise, it will contain the same text it held the last time you looked at it. The mode line will now describe the buffer and file in window two. It's hard to provide a mode line for each window, but making the mode line apply to the window you are in is the next best thing.

You can now edit in window two as you wish, while window one remains visible. When you are finished editing or looking at the text in window two, C-X 1 (^R One Window) returns to one-window mode. Window one expands to fill the whole screen, and window two disappears until the next C-X 2.

While you are in two window mode you can use C-X O (^R Other Window) to switch between the windows. After doing C-X 2, the cursor is in window two. Doing C-X O moves the cursor back to window one, to exactly where it was before the C-X 2. The difference between this and doing C-X 1 is that C-X O leaves window two visible on the screen. A second C-X O moves the cursor back into window two, to where it was before the first C-X O. And so on...

Often you will be editing one window while using the other just for reference. Then, the command C-M-V (^R Scroll Other Window) is very useful. It scrolls the other window without switching to it and switching back. It scrolls the same way C-V does: with no argument, a whole screen up; with an argument, that many lines up (or down, for a negative argument). With just a minus sign (no digits) as an argument, C-M-V scrolls a whole screenful backwards (what M-V does).

When you are finished using two windows, the C-X 1 command makes window two vanish. It doesn't

matter which window the cursor is in when you do the C-X 1; either way window two vanishes and window one remains. To make window one vanish and window two remain, give C-X 1 an argument.

The C-X 3 (^R View Two Windows) command is like C-X 2 but leaves the cursor in window one. That is, it makes window two appear at the bottom of the screen but leaves the cursor where it was. C-X 2 is equivalent to C-X 3 C-X O. C-X 3 is equivalent to C-X 2 C-X O, but C-X 3 is much faster.

Normally, the screen is divided evenly between the two windows. You can also redistribute the lines between the windows with the C-X ^ (^R Grow Window) command. It makes the currently selected window get one line bigger, or as many lines as is specified. With a negative argument, it makes the selected window smaller. The allocation of space to the windows is always remembered and changes only when you give a C-X ^ command.

After leaving two-window mode, you can still use C-X O, but the effect is slightly different. Window two does not appear, but whatever was being shown in it appears, in window one (the whole screen). Whatever buffer used to be in window one is stuck, invisibly, into window two. Another C-X O reverses the effect of the first. For example, if window one shows buffer B and window two shows buffer W2 (the usual case), and only window one is visible, then after a C-X O window one shows buffer W2 and window two shows buffer B.

16.1. Multiple Windows and Multiple Buffers

You can view one buffer in both windows. Give C-X 2 an argument as in C-U C-X 2 to go into window two viewing the same buffer as window one. Although the same buffer appears in both windows, they have different values of point, so you can move around in window two while window one continues to show the same text. Then, having found in window two the place you wish to refer to, you can go back to window one with C-X O to make your changes. Finally you can do C-X 1 to make window two leave the screen. If you are already in two window mode, C-U C-X O switches windows carrying the buffer from the old window to the new one so that both are viewing the same buffer.

Buffers can be selected independently in each window. The C-X B command selects a new buffer in whichever window the cursor is in. The other window's buffer does not change. When you do C-X 2 and window two appears it shows whatever buffer used to be visible in it when it was on the screen last.

If you have the same buffer in both windows, you must beware of trying to visit a different file in one of the windows with C-X C-V, because if you bring a new file into this buffer, it will replace the old file in *both* windows. To view different files in the two windows again, you must switch buffers in one of the windows first (with C-X B or C-X C-F, perhaps).

A convenient "combination" command for viewing something in the other window is C-X 4 (^R Visit in Other Window). With this command you can ask to see any specified buffer, file or tag in the other window. Follow the C-X 4 with either B and a buffer name, F or C-F and a file name, or T or "." and a tag name (See section 21 [TAGS], page 101.). This switches to the other window and finds there what you specified. If you were previously in one-window mode, two-window mode is entered.

17. Narrowing

C-X N	Narrow down to between point and mark.
C-X W	Widen to view the entire buffer.
C-X P	Narrow down to the page point is in.

"Narrowing" means focusing in on some portion of the buffer, making the rest temporarily invisible and inaccessible.

When you have narrowed down to a part of the buffer, that part appears to be all there is. You can't see the rest, you can't move into it (motion commands won't go outside the visible part), you can't change it in any way. However, it is not gone, and if you save the file you are editing all the invisible text will be saved. In addition to sometimes making it easier to concentrate on a single subroutine or paragraph by eliminating clutter, narrowing can be used to restrict the range of operation of a replace command.

The primary narrowing command is C-X N (^R Set Bounds Region). It sets the "virtual buffer boundaries" at point and the mark, so that only what was between them remains visible. Point moves to the top of the now-visible range, and the mark is left at the end, so that the region marked is unchanged.

The way to undo narrowing is to widen with C-X W (^R Set Bounds Full). This makes all text in the buffer accessible again.

Another way to narrow is to narrow to just one page, with C-X P (^R Set Bounds Page). See section 18 [Pages], page 79.

You can get information on what part of the buffer you are narrowed down to using the C-X = command. See section 11.4 [Filling], page 47.

Note that the virtual buffer boundaries are a powerful TECO mechanism used internally in EMACS in many ways. While only the commands described here set them so as you can see, many others set them temporarily using the TECO commands FS VB$ and FS VZ$, and restore them before finishing.

18. Commands for Manipulating Pages

C-M-L Insert formfeed.
C-X C-P Put point and mark around this page (or another page).
C-X [Move point to previous page boundary.
C-X] Move point to next page boundary.
C-X P Narrow down to just this (or next) page.
C-X L Count the lines in this page.
M-X What Page
Print current page and line number.

Files are often thought of as divided into pages by the ASCII character formfeed (↑L). For example, if a file is printed on a line printer, each page of the file, in this sense, will start on a new page of paper.

Most editors make the division of a file into pages extremely important. For example, they may be unable to show more than one page of the file at any time. EMACS treats a formfeed character just like any other character. It can be inserted with C-Q C-L (or, C-M-L), and deleted with Rubout. Thus, you are free to paginate your file, or not. However, since pages are often meaningful divisions of the file, commands are provided to move over them and operate on them. If you happen to like seeing only one page of the file at a time, you can use the PAGE library for that. See section 18.1 [PAGE], page 80.

The C-X [(^R Previous Page) command moves point to the previous page delimiter (actually, to right after it). If point starts out right after a page delimiter, it skips that one and stops at the previous one. A numeric argument serves as a repeat count. The C-X] (^R Next Page) command moves past the next page delimiter.

The comm and M-X What Page prints the page and line number of the cursor in the echo area. There is a separate command to print this information because it is likely to be slow and should not slow down anything else (The design of TECO is such that it is not possible to know the absolute number of the page you are in, except by scanning through the whole file counting pages).

The C-X C-P command (^R Mark Page) puts point at the beginning of the current page and the mark at the end. The page terminator at the end is included (the mark follows it). That at the front is excluded (point follows it). This command can be followed by a C-W to kill a page which is to be moved elsewhere.

A numeric argument to C-X C-P is used to specify which page to go to, relative to the current one. Zero means the current page. One means the next page, and -1 means the previous one.

The command C-X P (^R Set Bounds Page) narrows down to just one page. Everything before and after becomes temporarily invisible and inaccessible (See section 17 [Narrowing], page 77.). Use C-X W to undo this. Both page terminators, the preceding one and the following one, are excluded from the visible region.

Like C-X C-P, the C-X P command normally selects the current page, but allows you to specify which page explicitly relative to the current one with a numeric argument. However, when you are already narrowed down to one page, C-X P moves you to the next page (otherwise, it would be a useless no-op). One effect of this quirk is that several C-X P's in a row get first the current page and then successive pages.

Just what delimits pages is controlled by the variable Page Delimiter, which should contain a TECO search string (See section 19.3 [TECO search strings], page 85.) which will match a page separator. Normally, it contains a string containing just ↑L. For an INFO file, it might usefully be changed to ↑_↑L↑O↑_, which means that either a ↑_↑L or just a ↑_ (whatever separates INFO nodes) should be a page separator. In any case, page separators are recognized as such only at the beginning of a line. The paragraph commands consider each page boundary a paragraph boundary as well.

The C-X L command (^R Count Lines Page) is good for deciding where to break a page in two. It first prints (in the echo area) the total number of lines in the current page, and then divides it up into those preceding the current line and those following, as in

```
Page has 96 lines (72+25)
```

Notice that the sum is off by one; this is correct if point is not at the front of a line.

18.1. Editing Only One Page at a Time

The PAGE library is meant to allow the handling of pages as discrete, often independent units, letting you see only one page at a time, and providing commands to move between pages, split pages and join pages. It contrives to show the number of the page you are looking at in the mode line. You can also ask to see a "directory" of the pages in the file, or to insert it into the file. This is an extension of and replacement for the facility provided by the C-X P command in standard EMACS. It is an optional library because we do not think it is necessarily an improvement.

The commands in the PAGE library supplant and redefine commands in standard EMACS. Therefore, you cannot use them unless you give the command M-X Load Library↓PAGE<cr> explicitly. See section 22.2 [Libraries], page 112.

C-X]	Move to next page.
C-X [	Move to previous page.
C-X C-P	Move to page by absolute number.
C-X P	Split this page at point.
C-X J	Join this page to the next or previous one.
C-X W	See the whole file again.

The most fundamental thing to do with PAGE is to go to a specific page. This can be done by giving the page number as an argument to C-X C-P (^R Goto Page). If you give a number too big, the last page in the

file is selected.

For convenience, C-X C-P with no argument when you are looking at the whole file selects the page containing point. When you are looking at only one page, C-X C-P with no argument goes to the next page and with a negative argument goes to the previous page.

However, the main commands for moving forward or backward by pages are C-X [and C-X] (^R Goto Previous Page and ^R Goto Next Page). These take a numeric argument (either sign) and move that many pages.

When you want to go back to viewing the whole file instead of just one page, you can use the C-X W (^R Widen Bounds) command. These are the same characters that you would use in standard EMACS, but they run a different function that knows to remove the page number from the mode line.

The C-S (^R Incremental Search) and C-R (^R Reverse Search) are redefined to widen bounds first and narrow them again afterwards. So you can search through the whole file, but afterward see only the page in which the search ended. In fact, PAGE goes through some trouble to work with whatever search functions you prefer to use, and find them wherever you put them.

To split an existing page, you could insert a ↑L, but unless you do this while seeing the whole file, PAGE might get confused for a while. A way that is less tricky is to use C-X P (^R Insert Pagemark) which inserts the page mark, and narrows down to the second of the two pages formed from the old page. To get rid of a page mark without worry, use C-X J (^R Join Next Page). It gets rid of the page mark after the current page; or, with a negative argument, gets rid of the page mark before this page.

A page mark is defined as <CRLF>↑L, if you set the variable PAGE Flush CRLF to 1, a page mark is be <CRLF>↑L<CRLF>, which has the effect of making the CRLF at the beginning of each page invisible. This may be desirable for EMACS library source files. You can also specify some other string in place of ↑L; the value of Page Delimiter will be used. If Page Delimiter specifies multiple alternatives, the first alternative is the one PAGE will insert, but all will be recognized.

To see a list of all the pages in the file, each one represented by its first nonempty line, use M-X View Page Directory. It prints out the first non-blank line on each page, preceded by its page number. M-X Insert Page Directory inserts the same directory into the buffer at point. If you give it an argument, it tries to make the whole thing into a comment by putting the Comment Start string at the front of each line and the Comment End string at the end.

If the variable Page Setup Hook exists, PAGE will execute its value as the function for placing PAGE's functions on keys.

19. Replacement Commands

Global search-and-replace operations are not used as often in EMACS as they are in other editors, but they are still provided. In addition to the simple Replace operation which is like that found in most editors, there is a Query Replace operation which asks you, for each occurrence of the pattern, whether to replace it.

19.1. Query Replace

To replace every instance of FOO with BAR, you can do M-X Replace♦FOO♦BAR<cr>. Replacement is done only after point, so if you want to cover the whole buffer you must go to the beginning first. Unless the variable Case Replace is zero, an attempt is made to preserve case; give both FOO and BAR in lower case, and if a particular FOO is found with a capital initial or all capitalized, the BAR which replaces it will be likewise.

If you give Replace (or Query Replace) an argument, then it insists that the occurrences of FOO be delimited by break characters (or an end of the buffer). So you can find only the word FOO, and not FOO when it is part of FOOBAR.

To restrict the replacement to a subset of the buffer, set the region around it and type C-X N ("N" for "Narrow"). This makes all of the buffer outside that region temporarily invisible (but the commands that save your file still know that it is there!). Then do the replacement. Afterward, C-X W ("Widen") to make the rest of the buffer visible again. See section 17 [Narrowing], page 77.

If you are afraid that there may be some FOO's that should not be changed, EMACS can still help you. Use M-X Query Replace♦FOO♦BAR<cr>. This displays each FOO and waits for you to say whether to replace it with a BAR. The things you can type when you are shown a FOO are:

Space	to replace the FOO (preserving case, just like plain Replace, unless Case Replace is zero).
Rubout	to skip to the next FOO without replacing this one.
Comma	to replace this FOO and display the result. You are then asked for another input character, except that since the replacement has already been made, Rubout and Space are equivalent.
Altmode	to exit without doing any more replacements.
Period	to replace this FOO and then exit.
!	to replace all remaining FOO's without asking (Replace actually works by calling Query Replace and pretending that a ! was typed in).
^	to go back to the previous FOO (or, where it was), in case you have made a mistake. This works by jumping to the mark (Query Replace sets the mark each time it finds a FOO).
C-R	to enter a recursive editing level, in case the FOO needs to be edited rather than just

replaced with a BAR. When you are done, exit the recursive editing level with C-M-C.

C-W to delete the FOO, and then start editing the buffer. When you are finished editing whatever is to replace the FOO, exit the recursive editing level with C-M-C.

If you type any other character, the Query Replace is exited, and the character executed as a command. To restart the Query Replace, use C-X Altmode which is a command to re-execute the previous minibuffer command or extended command. See section 5 [M-X], page 19.

The first argument of Query Replace and Replace String is not simply a string; it is a kind of pattern, a TECO search string. See section 19.3 [TECO search strings], page 85.

19.1.1. Running Query Replace with the Minibuffer

Meta-% gives you a minibuffer pre-initialized with "MM Query Replace♦". This is the easiest way to invoke Query Replace. It also allows you to get Returns and Altmodes into the arguments.

With the minibuffer, Query Replace can be given a precomma argument, which says that the second string argument is actually a TECO program to be executed to perform the replacement, rather than simply a string to replace with.

When you invoke Query Replace from the minibuffer, the character C-] becomes special (because it is special in TECO programs). In order to get a C-] into the search string or the replacement string, you must use two of them. You can also use a C-] to quote an Altmode. In the minibuffer, Return has no syntactic significance, so there is no need for a way to quote it. However, in order to insert any control characters into the arguments, you need to quote them again with C-Q. So, to get C-Q C-X into the search string so as to search for a C-X, you have to type C-Q C-Q C-Q C-X.

19.2. Other Search-and-loop Functions

Here are some other functions related to replacement. Their arguments are TECO search strings (See section 19.3 [TECO search strings], page 85.). They all operate from point to the end of the buffer (or where C-X N stops them).

M-X Occur♦FOO<cr>
: which finds all occurrences of FOO after point. It prints each line containing one. With an argument, it prints that many lines before and after each occurrence.

M-X How Many♦FOO<cr>
: types the number of occurrences of FOO after point.

M-X Keep Lines♦FOO<cr>

kills all lines after point that don't contain FOO.

M-X Flush Lines♦FOO<cr>
kills all lines after point that contain FOO.

19.3. TECO Search Strings

The first string argument to Replace and Query Replace is actually a TECO search string. This means that the characters C-X, C-B, C-N, C-O, and C-Q have special meanings. C-X matches any character. C-B matches any "delimiter" character (anything which the word commands consider not part of a word, according to the syntax table. See section 22.4 [Syntax], page 115.). C-N negates what follows, so that C-N A matches anything but A, and C-N C-B matches any non-delimiter. C-O means "or", so that XYXY C-O *ZZZ* matches *either* XYXY or *ZZZ*. C-O can be used more than once in a pattern. C-Q quotes the following character, in case you want to search for one of the special control characters. However, you can't quote an Altmode or a Return in this way because its specialness is at an earlier stage of processing.

Some variables are supposed to have TECO search strings as their values. For example, Page Delimiter is supposed to be a search string to match anything which should start a page. This is so that you can use C-O to match several alternatives. In the values of such variables, C-B, C-N, C-O, C-Q, C-X and C-] are special, but Altmode is not. C-B through C-X are quoted with a C-Q, and C-] is quoted with another C-].

20. Editing Programs

Special features for editing programs include automatic indentation, comment alignment, parenthesis matching, and the ability to move over and kill balanced expressions. Many of these features are parameterized so that they can work for any programming language.

For each language there is a special "major mode" which customizes EMACS slightly to be better suited to editing programs written in that language. These modes sometimes offer special facilities as well.

See section 11.1 [Words], page 43. Moving over words is useful for editing programs as well as text.

See section 11.2 [Paragraphs], page 44. Most programming language major modes define paragraphs to be separated only by blank lines and page boundaries. This makes the paragraph commands useful for editing programs.

See section 21 [Tags], page 101. The TAGS package can remember all the labels or functions in a multi-file program and find any one of them quickly.

20.1. Major Modes

When EMACS starts up, it is in what is called "Fundamental mode", which means that the single and double character commands are defined so as to be convenient in general. More precisely, in Fundamental mode every EMACS option is set in its default state. For editing any specific type of text, such as Lisp code or English text, you can tell EMACS to change the meanings of a few commands to become more specifically adapted to the task. This is done by switching from Fundamental mode to one of the other major modes. Most commands remain unchanged; the ones which usually change are Tab, Rubout, and Linefeed. In addition, the commands which handle comments use the mode to determine how comments are to be delimited.

Selecting a new major mode is done with an M-X command. Each major mode is the name of the function to select that mode. Thus, you can enter Lisp mode by executing M-X Lisp (short for M-X Lisp Mode). The major modes are mutually exclusive; you can be in only one major mode at a time. When at top level, EMACS always says in the mode line which major mode you are in. You can specify which major mode should be used for editing a certain file by putting -*-<mode name>-*- somewhere in the first nonblank line of the file. For example, this file has -*-Text-*-.

Many major modes redefine the syntactical properties of characters appearing in the buffer. See section 22.4 [Syntax], page 115.

Most programming language major modes specify that only blank lines separate paragraphs. This is so that the paragraph commands do not become useless. They also cause Auto Fill mode to use the definition of Tab to indent the new lines it creates. This is because most lines are usually indented.

Major modes are standardly defined for the languages Lisp, Muddle, MIDAS, Macsyma, BCPL, BLISS, PASCAL, FORTRAN, TECO, and PL1.

There is also Text mode, designed for editing English text, or input to text justifier programs. See section 11 [Text], page 43.

20.2. Compiling Your Program

The command M-X Compile<cr> is used to compile the visited file. It knows how to compile it based on the major mode; for example, in MIDAS mode, it works by invoking MIDAS.

The first thing M-X Compile does is offer to save each buffer. This is because it is likely that all the buffers contain parts of the same program you are about to compile.

Then your program is compiled by executing a string of DDT commands. Normally, the commands are constructed from the visited file name and the major mode; for example, in MIDAS mode, the command

```
:MIDAS <filename>
```

would be used. When this is not sufficient, you can specify what to do by defining the variable Compile Command to be a TECO program to do the compilation. It could work by using the ↑K command to pass commands to DDT. The TECO program *must* use ↑\ to exit.

In some languages, all the files of a multi-file program must be compiled together, and the compiler must be given all the files, or one particular file which specifies all the others. EMACS libraries are an example of the first sort, and MIDAS programs an example of the second sort. In such cases, each of the files of the program could define a Compile Command with a local modes list which specifies all the files, or the appropriate one. See the file AI:EMACS1;CCL. > for an example of doing this for an EMACS library.

Major modes which are not known to M-X Compile can work with it by setting Compile Command. In this case the value of Compile Command must be independent of the name of the file. When it is executed, it can find the filename to use in q-register 1.

20.3. Indentation Commands for Code

Tab	Indents current line.
Linefeed	Equivalent to Return followed by Tab.
M-^	Joins two lines, leaving one space between if appropriate.
M-\	Deletes all spaces and tabs around point.
M-M	Moves to the first nonblank character on the line.

Most programming languages have some indentation convention. For Lisp code, lines are indented according to their nesting in parentheses. For assembler code, almost all lines start with a single tab, but some have one or more spaces as well. Indenting TECO code is an art rather than a science, but it is often useful to indent a line under the previous one.

The way to request indentation is with the Tab command. Each major mode defines this command to perform the sort of indentation appropriate for the particular language. In Lisp mode, Tab aligns the line according to its depth in parentheses. No matter where in the line you are when you type Tab, it aligns the line as a whole. In MIDAS mode, Tab inserts a tab, that being the standard indentation for assembly code. In TECO mode, Tab realigns the current line to match a previous line. PL1 mode (See the file INFO; EPL1 >.) knows in great detail about the keywords of the language so as to indent lines according to the nesting structure.

The command Linefeed (^R Indent New Line) does a Return and then does a Tab on the next line. Thus, Linefeed at the end of the line makes a following blank line and supplies it with the usual amount of indentation, just as Return would make an empty line. Linefeed in the middle of a line breaks the line and supplies the usual indentation in front of the new line.

The inverse of Linefeed is Meta-^ or C-M-^ (^R Delete Indentation). This command deletes the indentation at the front of the current line, and the line boundary as well. They are replaced by a single space, or by no space if before a ")" or after a "(", or at the beginning of a line. To delete just the indentation of a line, go to the beginning of the line and use Meta-\ (^R Delete Horizontal Space), which deletes all spaces and tabs around the cursor.

To insert an indented line before the current one, do C-A, C-O, and then Tab. To make an indented line after the current one, use C-E Linefeed.

To move over the indentation on a line, do Meta-M or C-M-M (^R Back to Indentation). These commands, given anywhere on a line, position the cursor at the first nonblank character on the line.

20.4. Automatic Display Of Matching Parentheses

The purpose of the EMACS parenthesis-matching feature is to show automatically how parentheses balance in text being typed in. When this feature is enabled, after a close parenthesis or other close bracket character is inserted the cursor automatically moves for an instant to the open which balances the newly inserted character. The cursor stays at the open parenthesis for a second before returning home, if you don't type any more commands during that time. If you type more commands before the second is up, EMACS won't wait the whole second.

It is worth emphasizing that the real location of the cursor, the place where your type-in will be inserted, is not affected by the close parenthesis matching feature. It stays after the close parenthesis, where it would normally be. Only the spot on the screen moves away and back. You can type ahead freely as if the matching feature were not there. In fact, if you type fast enough, you won't see the cursor move. You must pause after typing a close parenthesis to see the open parenthesis.

The variable Display Matching Paren controls parenthesis display. If it is zero, the feature is disabled. If the variable is nonzero, then its absolute value is the number of seconds for the cursor to stay at the open parenthesis before coming back to its real location. The sign of the variable is also significant: if it is negative, then the open parenthesis is shown only if it is already on the screen. If the variable is positive, then EMACS will actually recenter the window to show the text around the open parenthesis. The default setting of the variable is -1.

An additional parameter is whether EMACS should warn you by ringing the bell if you type an unmatched close parenthesis. The default is to warn you if you are editing a language in which parentheses are essential, like Lisp, but not to do so for languages in which parentheses are not so crucial. This is controlled by the variable Permit Unmatched Paren. When it is 1, you are never warned (they are always "permitted"). When it is -1, you are warned only in Lisp mode and similar modes (this is the default). Note that these modes operate by locally setting the variable to 1 if it was -1. When it is 0, you are warned regardless of the major mode. Unmatched parens are *always* "permitted" in that EMACS will never refuse to insert them.

While this feature was intended primarily for Lisp, it can be used just as well for any other language, and it is not dependent on what major mode you are in. It is expected that you wouldn't want it in Text mode, so Text mode sets the variable Display Matching Paren locally to zero. If you do want the feature in Text mode, you can create a Text Mode Hook variable which sets the variable back to -1. See the file INFO;CONV >, mode Hooks, for more info on Text Mode Hook. The way to control which characters trigger this feature is with the syntax table. Any character whose Lisp syntax is ")" will cause the matching character with syntax "(" to be shown. Most major modes automatically set up the syntax table (See section 22.4 [Syntax],

page 115.).

The syntax table also controls what is done with the case of "mismatched" parens, as in "[ )". The third slot in a close parenthesis character's syntax table entry should be the proper matching open parenthesis character, if you want this feature turned on. If that slot contains a space instead, then any open parenthesis character is considered a legitimate match.

The implementation of this feature uses the TECO flag FS ^R PAREN♦. See section 22.5 [FS Flags], page 117.

20.5. Manipulating Comments

The comment commands insert, kill and align comments. There are also commands for moving through existing code and inserting comments.

C-;	Insert or align comment.
M-;	The same.
C-M-;	Kill comment.
C-X ;	Set comment column.
M-N	Move to Next line and insert comment.
M-P	Move to Previous line and insert comment.
M-J	Continue a comment on a new line.
M-Linefeed	The same.

The command that creates a comment is Control-; or Meta-; (^R Indent for Comment). It moves to the end of the line, indents to the comment column, and inserts whatever string EMACS believes comments are supposed to start with (normally ";"). If the line goes past the comment column, then the indentation is done to a suitable boundary (usually, a multiple of 8). If the language you are editing requires a terminator for comments (other than the end of the line), the terminator is inserted too, but point goes between the starter and the terminator.

Control-; can also be used to align an existing comment. If a line already contains the string that starts comments, then C-; just moves point after it and indents it to the right place (where a comment would have been created if there had been none).

Even when an existing comment is properly aligned, C-; is still useful for moving directly to the start of the comment.

C-M-; (^R Kill Comment) kills the comment on the current line, if there is one. The indentation before the start of the comment is killed as well. If there does not appear to be a comment in the line, nothing is done. Since killed text can be reinserted with C-Y, this command is useful for moving a comment from one

line to another.

20.5.1. Multiple Lines of Comments

If you wish to align a large number of comments, you can give Control-; an argument and it indents what comments exist on that many lines, creating none. Point is left after the last line processed (unlike the no-argument case).

When adding comments to a long stretch of existing code, the commands M-N (^R Down Comment Line) and M-P (^R Up Comment Line) may be useful. They are like C-N and C-P except that they do a C-; automatically on each line as you move to it, and delete any empty comment from the line as you leave it. Thus, you can use M-N to move down through the code, putting text into the comments when you want to, and allowing the comments that you don't fill in to be removed because they remained empty.

If you are typing a comment and find that you wish to continue it on another line, you can use the command Meta-J or Meta-Linefeed (^R Indent New Comment Line), which terminates the comment you are typing, creates or gobbles a new blank line, and begins a new comment indented under the old one. When Auto Fill mode is on, going past the fill column while typing a comment causes the comment to be continued in just this fashion. Note that if the next line is not blank, a blank line is created, instead of putting the next line of the comment on the next line of code. To do that, use M-N.

20.5.2. Double and Triple Semicolons in Lisp

In Lisp code there are conventions for comments which start with more than one semicolon. Comments which start with two semicolons are indented as if they were lines of code, instead of at the comment column. Comments which start with three semicolons are supposed to start at the left margin. EMACS understands these conventions by indenting a double-semicolon comment using Tab, and by not changing the indentation of a triple-semicolon comment at all. (Actually, this rule applies whenever the comment starter is a single character and is duplicated). Note that the :@ program considers a four-semicolon comment a subtitle in Lisp code.

20.5.3. Options Controlling Comments

The comment column is stored in the variable Comment Column. You can set it to a number explicitly. Alternatively, the command C-X ; (^R Set Comment Column) sets the comment column to the column point is at. C-U C-X ; sets the comment column to match the last comment before point in the buffer, and then does a Meta-; to align the current line's comment under the previous one.

Many major modes supply default local values for the comment column. In addition, C-X ; automatically makes the variable Comment Column local. Otherwise, if you change the variable itself, it changes globally (for all buffers) unless it has been made local in the selected one. See section 22.6 [Locals], page 118.

The string recognized as the start of a comment is stored in the variable Comment Start, while the string used to start a new comment is kept in Comment Begin (if that is zero, Comment Start is used for new comments). This makes it possible for you to have any ";" recognized as starting a comment but have new comments begin with ";; ** ".

The string used to end a comment is kept in the variable Comment End. In many languages no comment end is needed as the comment extends to the end of the line. Then, this variable is a null string.

20.6. Lisp Mode and Muddle Mode

Lisp's simple syntax makes it much easier for an editor to understand, as a result, EMACS can do more for Lisp, and with less work, than for any other language.

Lisp programs should be edited in Lisp mode. In this mode, Tab is defined to indent the current line according to the conventions of Lisp programming style. It does not matter where in the line Tab is used; the effect on the line is the same. The function which does the work is called ^R Indent for Lisp. Linefeed, as usual, does a Return and a Tab, so it moves to the next line and indents it.

As in most modes where indentation is likely to vary from line to line, Rubout is redefined to treat a tab as if it were the equivalent number of space (^R Backward Delete Hacking Tabs). This makes it possible to rub out indentation one position at a time without worrying whether it is made up of spaces or tabs. Control-Rubout does the ordinary type of rubbing out which rubs out a whole tab at once.

Paragraphs are defined to start only with blank lines so that the paragraph commands can be useful. Auto fill indents the new lines which it creates. Comments start with ";". In Lisp mode, the action of the word-motion commands is affected by whether you are in atom word mode or not.

The LEDIT library allows EMACS and Lisp to communicate, telling Lisp the new definitions of functions which you edit in EMACS. See the file INFO;LEDIT >.

The language Muddle is a variant form of Lisp which shares the concept of using parentheses (of various sorts) as the main syntactical construct. It can be edited using Muddle mode, which is almost the same as Lisp mode and provides the same features, differing only in the syntax table used.

20.6.1. Moving Over and Killing Lists and S-expressions

C-M-F	Move Forward over s-expression.
C-M-B	Move Backward.
C-M-K	Kill s-expression forward.
C-M-Rubout	Kill s-expression backward.
C-M-U	Move Up and backward in list structure.
C-M-(	The same.
C-M-)	Move up and forward in list structure.
C-M-D	Move Down and forward in list structure.
C-M-N	Move forward over a list.
C-M-P	Move backward over a list.
C-M-T	Transpose s-expressions.
C-M-@	Put mark after s-expression.
M-(	Put parentheses around next s-expression(s).
M-)	Move past next close parenthesis and reindent.

By convention, EMACS commands that deal with balanced parentheses are usually Control-Meta-characters. They tend to be analogous in function to their Control- and Meta- equivalents. These commands are usually thought of as pertaining to Lisp, but can be useful with any language in which some sort of parentheses exist (including English).

To move forward over an s-expression, use C-M-F (^R Forward Sexp). If the first non-"useless" character after point is an "(", C-M-F moves past the matching ")". If the first character is a ")", C-M-F just moves past it. If the character begins an atom, C-M-F moves to the atom-break character that ends the atom. C-M-F with an argument repeats that operation the specified number of times; with a negative argument, it moves backward instead.

The command C-M-B (^R Backward Sexp) moves backward over an s-expression; it is like C-M-F with the argument negated. If there are "'"-like characters in front of the s-expression moved over, they are moved over as well. Thus, with point after " 'FOO ", C-M-B leaves point before the "'", not before the "F".

These two commands (and most of the commands in this section) do not know how to deal with the presence of comments. Although that would be easy to fix for forward motion, for backward motion the syntax of Lisp makes it nearly impossible. Comments by themselves wouldn't be so bad, but handling comments and "|" both is impossible to do locally. In a line " ((FOO ; | BAR ", are the open parentheses inside of a "| ... |" atom? I do not think it would be advisable to make C-M-F handle comments without making C-M-B handle them as well.

For this reason, two other commands which move over lists instead of s-expressions are often useful. They are C-M-N (^R Forward List) and C-M-P (^R Backward List). They act like C-M-F and C-M-B except that they don't stop on atoms; after moving over an atom, they move over the next expression, stopping after

moving over a list. With this command, you can avoid stopping after all of the words in a comment.

Killing an s-expression at a time can be done with C-M-K and C-M-Rubout (^R Forward Kill Sexp and ^R Backward Kill Sexp). C-M-K kills the characters that C-M-F would move over, and C-M-Rubout kills what C-M-B would move over.

C-M-F and C-M-B stay at the same level in parentheses, when that's possible. To move *up* one (or n) levels, use C-M-(or C-M-) [^R Backward Up List and ^R Forward Up List]. C-M-(moves backwards up past one containing "(". C-M-) moves forwards up past one containing ")". Given a positive argument, these commands move up the specified number of levels of parentheses. C-M-U is another name for C-M-(, which is easier to type, especially on non-Meta keyboards. If you use that name, it is useful to know that a negative argument makes the command move up forwards, like C-M-).

To move *down* in list structure, use C-M-D (^R Down List). It is nearly the same as searching for a "(".

A somewhat random-sounding command which is nevertheless easy to use is C-M-T (^R Transpose Sexps), which moves the cursor forward over one s-expression, dragging the previous s-expression along. An argument serves as a repeat count, and a negative argument drags backwards (thus canceling out the effect of a positive argument). An argument of zero, rather than doing nothing, transposes the s-expressions at the point and the mark

To make the region be the next s-expression in the buffer, use or C-M-@ (^R Mark Sexp) which sets mark at the same place that C-M-F would move to. C-M-@ takes arguments like C-M-F. In particular, a negative argument is useful for putting the mark at the beginning of the previous s-expression.

The commands M-([^R Insert ()] and M-) [^R Move Over)] are designed for a style of editing which keeps parentheses balanced at all times. M-(inserts a pair of parentheses, either together as in "()", or, if given an argument, around the next several s-expressions, and leaves point after the open parenthesis. Instead of typing "(FOO)", you can type M-(FOO, which has the same effect except for leaving the cursor before the close parenthesis. Then you type M-), which moves past the close parenthesis, deleting any indentation preceding it (in this example there is none), and indenting with Linefeed after it.

The list commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to act like an open parenthesis. See section 22.4 [Syntax], page 115.

20.6.2. Commands for Manipulating Defuns

C-M-[, C-M-A	Move to beginning of defun.
C-M-], C-M-E	Move to end of defun.
C-M-H	Put region around whole defun.

In EMACS, a list at the top level in the buffer is called a defun, regardless of what function is actually called, because such lists usually call defun. There are EMACS commands to move to the beginning or end of the current defun: C-M-[(^R Beginning of Defun) moves to the beginning, and C-M-] (^R End of Defun) moves to the end. If you wish to operate on the current defun, use C-M-H (^R Mark Defun) which puts point at the beginning and mark at the end of the current or next defun. Alternate names for these two commands are C-M-A for C-M-[and C-M-E for C-M-]. The alternate names are easier to type on many non-Meta keyboards.

20.7. Lisp Grinding

The best way to keep Lisp code indented properly is to use EMACS to re-indent it when it is changed. EMACS has commands to indent properly either a single line, a specified number of lines, or all of the lines inside a single s-expression.

Tab	In Lisp mode, reindents line according to parenthesis depth.
Linefeed	Equivalent to Return followed by Tab.
M-^	Join two lines, leaving one space between them if appropriate.
C-M-Q	Reindent all the lines within one list.
C-M-G	Grind a list, moving code between lines.

The basic indentation function is ^R Indent for Lisp, which gives the current line the correct indentation as determined from the previous lines' indentation and parenthesis structure. This function is normally found on C-M-Tab, but when in Lisp mode it is placed on Tab as well (Use Meta-Tab to insert a tab). When given at the beginning of a line, it leaves point after the indentation; when given inside the text on the line, point remains fixed with respect to the characters around it.

When entering a large amount of new code, it becomes useful that Linefeed (^R Indent New Line) is equivalent to a Return followed by a Tab. In Lisp mode, a Linefeed creates or moves down onto a blank line, and then give it the appropriate indentation.

To join two lines together, use the Meta-^ or Control-Meta-^ command (^R Delete Indentation), which is approximately the opposite of Linefeed. It deletes any spaces and tabs at the front of the current line, and then deletes the line separator before the line. A single space is then inserted, if EMACS thinks that one is needed there. Spaces are not needed before a close parenthesis, or after an open parenthesis.

If you are dissatisfied about where Tab wants to place the second and later lines of an s-expression, you can override it. If you alter the indentation of one of the lines yourself, then Tab will indent successive lines of the same list to be underneath it. This is the right thing for functions which Tab indents unaesthetically. Of course, it is the wrong thing for PROG tags (if you like to un-indent them), but it's impossible to be right for both.

When you wish to re-indent code which has been altered or moved to a different level in the list structure, you have several commands available. You can re-indent a specific number of lines by giving the ordinary indent command (Tab, in Lisp mode) an argument. This indents as many lines as you say and moves to the line following them. Thus, if you underestimate, you can repeat the process later.

You can re-indent the contents of a single s-expression by positioning point before the beginning of it and typing Control-Meta-Q (^R Indent Sexp). The line the s-expression starts on is not re-indented; thus, only the relative indentation with in the s-expression, and not its position, is changed. To correct the position as well, type a Tab before the C-M-Q.

Another way to specify the range to be re-indented is with point and mark. The command C-M-\ (^R Indent Region) applies Tab to every line whose first character is between point and mark. In Lisp mode, this does a Lisp indent.

A more powerful grind command which can move text between lines is C-M-G (^R Format Code). You might or might not like it. It knows in different ways about Lisp code and Macsyma code.

20.8. Editing Assembly-Language Programs

M-X MIDAS Mode is designed for editing programs written in MIDAS or other PDP-10 or PDP-11 assemblers. In MIDAS mode, comments start with ";", and "<" and ">" have the syntax of parentheses. In addition, there are five special commands which understand the syntax of instructions and labels. These commands are:

C-M-N	Go to Next label.
C-M-P	Go to Previous label.
C-M-A	Go to Accumulator field of instruction.
C-M-E	Go to Effective Address field.
C-M-D	Kill next word and its Delimiting character.
M-[	Move up to previous paragraph boundary.
M-]	Move down to next paragraph boundary.

Any line which is not indented and is not just a comment is taken to contain a label. The label is everything up to the first whitespace (or the end of the line). C-M-N (^R Go to Next Label) and C-M-P (^R Go to Previous Label) both position the cursor right at the end of a label; C-M-N moves forward or down and C-M-P moves backward or up. At the beginning of a line containing a label, C-M-N moves past it. Past the label on the same line, C-M-P moves back to the end of it. If you kill a couple of indented lines and want to insert them right after a label, these commands put you at just the right place.

C-M-A (^R Go to AC Field) and C-M-E (^R Go to Address Field) move to the beginning of the accumulator (AC) or effective address fields of a PDP-10 instruction. They always stay on the same line,

moving either forward or backward as appropriate. If the instruction contains no AC field, C-M-A positions to the start of the address field. If the instruction is just an opcode with no AC field or address field, a space is inserted after the opcode and the cursor left after the space. In PDP-11 programs, C-M-A moves to the first operand and C-M-E moves to the second operand.

Once you've gone to the beginning of the AC field you can often use C-M-D (^R Kill terminated Word) to kill the AC name and the comma which terminates it. You can also use it at the beginning of a line, to kill a label and its colon, or after a line's indentation to kill the opcode and the following space. This is very convenient for moving a label from one line to another. In general, C-M-D is equivalent to M-D C-D, except that all the characters are saved on the kill ring, together. C-D, a "deletion" command, doesn't save on the kill ring if not given an argument.

The M-[and M-] commands are not, strictly speaking, redefined by MIDAS mode, since they always go up or down to a paragraph boundary. However, in MIDAS mode the criterion for a paragraph boundary is changed by setting the variable Paragraph Delimiter (See section 11.2 [Paragraphs], page 44.) so that only blank lines (and pages) delimit paragraphs. So, M-[moves up to the previous blank line and M-] moves to the next one.

20.9. Major Modes for Other Languages

MACSYMA mode redefines the syntax of words and s-expressions in an attempt to make it easier to move over MACSYMA syntactic units. In addition, the C-M-G "grind" command is told to grind text as MACSYMA instead of as Lisp. Also, the syntax of MACSYMA comments is understood.

TECO mode is good for editing EMACS library source files. It makes Tab be ^R Indent Nested (see its self-documentation). Comments start with "!* " and end with "!". In addition, the PURIFY library which contains many things useful for processing library sources (including the commands to compile them) is loaded. M-' and M-" are connected to functions ^R Forward TECO Conditional and ^R Backward TECO Conditional which move forward and backward over balanced TECO conditionals. In TECO mode on a terminal with a Meta key, it may be useful to set the TECO flag FS CTLMTA♦ which causes Control-Meta commands to insert Control characters. See section 22.5 [FS Flags], page 117.

PL1 mode is for editing PL1 code, and causes Tab to indent an amount based on the previous statement type. The body of the implementation of PL1 mode is in the library PL1, which is loaded automatically when necessary. See the file INFO;EPL1 >.

PASCAL mode is similar to PL1 mode, for PASCAL. It is in the library called PASCAL. See the file INFO;EPASC >.

There are also modes for BLISS, BCPL and FORTRAN, but no documentation for them except that in the libraries themselves. Any volunteers?

21. The TAGS Package.

The TAGS package remembers the locations of the function definitions in a file and enables you to go directly to the definition of any function, without searching the whole file.

The functions of several files that make up one program can all be remembered together if you wish; then the TAGS package will automatically select the appropriate file as well.

21.1. How to Make a Tags File for a Program

To use the TAGS package, you must create a tag table for the text file or files in your package. Normally, the tag table does not reside in any of those files, but in a separate tag table file which contains the names of the text files which it describes. Tag tables are generated by the :TAGS program. The same program can be used to update the tags file if it becomes very far out of date (slight inaccuracies do not matter). Tag tables for INFO files work differently; the INFO file contains its own tag table, which describes only that file. See section 21.8 [INFO], page 108, for how to deal with them.

The normal mode of operation of the :TAGS program is to read in an existing tags file and update it by rescanning the source files that it describes. The old tag table file itself tells :TAGS which source files to process. When making a new tag table you must start by making a skeleton. Then :TAGS is used to turn the skeleton into an accurate tag table.

A skeleton tag table is like a real one except that it is empty; there are no tags in it. It contains exactly this much data, for each source file that it is going to describe:

```
<filenames>
0,<language>
↑_
```

The languages that :TAGS understands now are TECO, LISP, MIDAS, FAIL, PALX, MUDDLE, MACSYMA, TJ6, and R. MIDAS will do for MACRO-10 files. Any incompletely specified filenames will default to > and to the directory on which the tags file itself is stored. The "0," *must* be present, since :TAGS expects that there will be a number in that place and will be completely confused if there is not. The CRLF after each ↑_ also *must* be present. You can omit both the last ↑_ and its CRLF together, however.

Thus, a skeleton tags file for the files EMACS;USRCOM > and EMACS;TAGS > would look like

```
EMACS;USRCOM >
0,TECO
↑_
EMACS;TAGS >
0,TECO
↑_
```

If this were written out as EMACS;EMACS TAGS, you could then do

```
:TAGS EMACS;EMACS    (default FN2 is TAGS)
```

which would tell :TAGS to read in the tags file, and write back an up-to-date tags file for the same set of source files. To update the tags file because lots of changes have been made, the same command to the :TAGS program will work. See section 21.6 [Edit], page 106, for info on adding, deleting, or renaming files in existing tags files.

21.2. How to Tell EMACS You Want to Use TAGS

Before you can use the TAGS package, you must tell EMACS the name of the tags file you want to use. This is done with the command

```
M-X Visit Tag Table↓ <filenames> <cr>
```

The FN2 of "TAGS" need not be mentioned.

EMACS can only know about one tag table file at a time, so doing a second M-X Visit Tag Table causes the first one to be forgotten (or written back if you have added definitions to it).

Giving M-X Visit Tag Table a nonzero numeric argument, as in

```
1 M-X Visit Tag Table↓ <filenames> <cr>
```

has the additional effect of setting the variable Tags Find File nonzero, which causes the TAGS package to use Find File rather than Visit File when it needs to switch files. This causes all the files to remain resident in the EMACS, in different buffers. In the default mode, visiting a tag in a different file read it in on top of the old file, in the same buffer (but it offers to write out changes if there are any). Warning: you can easily run out of address space by making too many buffers, this way.

Visit Tag Table is essentially equivalent to selecting the buffer "*TAGS*" and visiting the tag table file in that buffer, then returning to the previously selected buffer. Afterwards, M-X List Buffers will show the buffer *TAGS* visiting that file. The only difference is that M-X Visit Tag Table causes the out of core portions of the TAGS package to be loaded.

21.3. Jumping to a Tag

To jump to the definition of a function, use the command Meta-Period <tag name> <cr>. You will go straight to the definition of the tag. If the definition is in a different file then TAGS will visit that file. If it is in the same file, TAGS will leave the mark behind and print "^@" in the echo area.

If Meta-Period is used before M-X Visit Tag Table has been done, it will ask for the name of a tag table file. After you type this name and a <cr>, you type the name of the tag as usual.

You do not need to type the complete name of the function; any substring will do. But this implies that sometimes you won't get the function you intended. When that happens, C-U Meta-Period will find the "next" function matching what you typed (next, in the order of listing in the tag table). Thus, if you wanted to find the definition of X-SET-TYPE-1 and you said just TYPE-1, you might find X-READ-TYPE-1 instead. You could then type C-U Meta-Period's until you reached X-SET-TYPE-1.

If you want to make sure you reach a precise function the first time, you should just include a character of context before and after its name. Thus, in a Lisp program, put a space before and after the function name. In a MIDAS program, put a linefeed before it and a colon after.

21.4. Other Operations on Tag Tables

21.4.1. Adding a New Function to a Tag Table

When you define a new function, its location doesn't go in the tag table automatically. That's because EMACS can't tell that you have defined a function unless you issue the command to say so by invoking the function ^R Add Tag. Since the operation of adding a tag to a tag table has proved not to be very necessary, this function no longer placed on any character, by default. You can invoke with M-X or place on a key if you like. From this section, let's assume you have placed it on C-X Period.

When you type the command C-X Period, the pointer should be on the line that introduces the function definition, after the function name and the punctuation that ends it. Thus, in a Lisp program, you might type "(DEFUN FOO " (note the space after FOO) and then type the C-X Period. In a MIDAS program, you might give the C-X Period after typing "FOO:". In a TECO program in EMACS format, you might type C-X Period after "!Set New Foo:!".

C-X Period modifies only the copy of the tag table loaded into EMACS. To modify the tag table file itself, you must cause it to be written out. Do this by selecting the buffer *TAGS* and saving it with C-X C-S, or with M-X Save All Files. There is also a function M-X ? Save Tag Table for doing it.

Although local modifications to a file do not degrade the efficiency of the TAGS package or require that the tag table be updated with :TAGS, moving a function a great distance make make it much slower to find that function. In this case, you can "add" the function to the tag table with C-X Period to give the table its new location. Or you can just run :TAGS again to update everything, as is usually done.

21.4.2. How to Process All the Files in a Tag Table

The TAGS package contains a function M-X Next File which visits, one by one, all the files described by the selected tag table. This is useful when there is something to be done to all of the files in the package. To start off the sequence, do C-U 1 M-X Next File, which visits the first file. When you are finished operating on one file, do M-X Next File (no argument) to see the next. When all the files have been processed, M-X Next File♦ gives an error.

The files of the package are visited in the order that they are mentioned in the tag table, and the current place in the sequence is remembered by the pointer in the buffer *TAGS* which holds the tag table. Thus, if you visit a tag in a different file in the middle of a M-X Next File sequence, you will screw it up unless you return to the proper file again by visiting a tag (or go into the buffer *TAGS* and reset the pointer). However, visiting any other files directly, not using TAGS, does not interfere with the sequence, and the next M-X Next File will go just where it would have gone.

Next File is also useful as a subroutine in functions that wish to perform an automatic transformation (such as a Query Replace) on each file. Such functions should call Next File with a precomma argument as in 1,M(M.M Next File♦) or 1,1M(M.M Next File♦). The precomma argument tells Next File to return 0 instead of giving an error when there are no more files to process. Normally, it returns -1.

Here is an example of TECO code to do a Query Replace on all of the files listed in the visited tag table:

```
1M(M.M Next File♦)
< M(M.M Query Replace♦)FOO♦BAR♦
  1,M(M.M Next File♦);>
```

Tags Search and Tags Query Replace (see below) both work using Next File.

21.4.3. Multi-File Searches and Replacements

The TAGS package contains a function Tags Search which will search through all of the files listed in the visited tag table in the order they are listed. Do M-X Tags Search♦<string><cr> to find every occurrence of <string>. <string> is a TECO search string in which special TECO search characters such as ↑O, ↑X, ↑N, ↑B, and ↑Q are allowed. See section 19.3 [TECO Search Strings], page 85.

When M-X Tags Search reaches the end of the buffer, it visits the next file automatically, typing its name in the echo area. As soon as M-X Tags Search finds one occurrence, it returns. But it defines the command Control-Period to resume the search from wherever point is.

Warning: use of Tags Search after setting Tags Find File to 1 can create more buffers than EMACS can handle. This results in an URK "Running out of core" error. After the error, you might be at TECO command level, outside of EMACS. If your type-in is echoed at the bottom of the screen, this has happened. You should immediately type MM Kill Some Buffers♦♦, kill some, and then do :M..I.♦♦ to reenter EMACS.

M-X Tags Query Replace does a Query Replace over all the files in a tag table. Like M-X Tags Search, it sets Control-. up to be a command to continue the Query Replace, in case you wish to exit, do some editing, and then resume scanning.

The library MQREPL enables you to use Next File to repeat a sequence of many Query Replace commands over a set of files, performing all the replacements on one file at a time.

21.4.4. Miscellaneous Applications of Tags

M-X List Tags♦<file><cr> lists all the tags in the specified file. Actually, all the files in the tag table whose names contain the string <file> are listed.

M-X Tags Apropos♦<pat><cr> lists all known tags whose names contain <pat>.

M-X Tags File List inserts in the buffer a list of the files known in the visited tag table.

M-X Tags Rescan runs :TAGS over the visited tag table and revisits it. This is the most convenient way to update the tag table.

M-X View Arglist♦<tag><cr> lets you look briefly at the line on which a tag is defined, and at the lines of comments which precede the definition. This is a good way to find out what arguments a function needs. The file is always loaded into a separate buffer, when this command is used.

M-X What Tag? tells you which function's definition you are in. It looks through the tag table for the tag which most nearly precedes point.

21.5. What Constitutes a Tag

In Lisp code, a function definition must start with an "(" at the beginning of a line, followed immediately with an atom which starts with "DEF" (and does not start with "DEFP"), or which starts with "MACRO", or

which starts with "ENDF". The next atom on the line is the name of the tag. If there is no second atom on the line, there is no tag.

In MIDAS code, a tag is any symbol that occurs at the beginning of a line and is terminated with a colon or an equal sign. MIDAS mode is good for MACRO-10 also.

FAIL code is like MIDAS code, except that one or two ↓'s or "^"'s are allowed before a tag, and spaces are allowed between the tag name and the colon or equal sign, and _ is recognized as equivalent to =.

PALX code is like MIDAS code, except that spaces are allowed between a tag and the following colon or equals, and local tags such as "10$" are ignored.

In TECO code, a tag starts with an "!" and ends with a ":!". There may be any number of tags on a line, but the first one must start at the beginning of a line.

In MUDDLE code, a tag is identified by a line that starts with "<DEFINE " or "<DEFMAC ", followed by a symbol.

In MACSYMA code, a function definition is recognized when there is a symbol at the beginning of a line, terminated with a "(" or "[", and there is a ":" later on in the line. If the symbol itself is terminated with a ":", a variable definition is recognized.

In R text, any line which starts with ".de" or ".am" or ".rtag" defines a tag. The name of the tag is what follows, up to the second run of spaces or the end of the line. There is no ".rtag" in R; define it to be a null macro, if you like, and use it to put in tags for chapters, or anything else. Any macro whose name starts with "de" or "am" or "rtag", such as ".define" or ".amplify", also defines a tag.

In TJ6 text, any line which starts with ".C TAG" starts a tag. The name of the tag is whatever follows the spaces which should follow the "C TAG", up to the next space or the end of the line.

21.6. Adding or Removing Source Files

A tag table file is a sequence of entries, one per file. Each entry looks like

```
<filenames>
<count>,<language>
<data lines>
^_
```

<filenames> are the fully defaulted names of the file, <language> is one of the languages that TAGS knows how to process, and <data lines> are the actual tag information (described below). The CRLF after each ^_ *must* be present. You can omit both the last ^_ and its CRLF together, however.

A tags file is for the most part an ordinary ASCII file, and any changes you make in it, including changes to the source files' names, will do what they appear to do.

The one exception is that each entry contains a count, in decimal, of the number of characters in it, including the ^_ and CRLF. If you edit the contents of an individual source file's entry, and change its length, then the tags file is no good for use in editing until you run :TAGS over it. :TAGS ignores the specified count and always writes the correct count. If you are sure that the length is unchanged, or if you change the count manually, then running :TAGS is not necessary, but you do so at your own risk. If you screw things up, use :TAGS to fix the file.

Thus, if you are changing a source file's name, you should simply change the name where it is present in the tags file, and run :TAGS over it if necessary.

To add a new source file, simply insert a dummy entry of the sort used in making a new tags file. Then use :TAGS to turn it into a real entry. Unless you go to the trouble of putting a valid count in the dummy entry, you must run :TAGS again before using the file.

You can delete a source file from a tags file by deleting its entire entry. You can also change the order of the entries without doing any harm (the order of the entries doesn't matter very often). Since the counts of the remaining entries are still valid, you need not run :TAGS over the file again.

You can edit everything else in the tags file too, if you want to. You might want to change a language name once in a while, but I doubt you will frequently want to add or remove tags, especially since that would all be undone by the next use of :TAGS!

21.7. How a Tag Is Described in the Tag Table

A tag table file consists of one or more subunits in succession. Each subunit lists the tags of one source file. Each subunit has the overall format described in the previous section, containing zero or more lines describing tags. Here we give the format of each of those lines.

Starting with the third line of the tag table entry, each line describes a tag. It starts with a copy of the beginning of the line that the tag is defined on, up through the tag name and its terminating punctuation. Then there is a rubout, followed by the character position in decimal of the place in the line where copying stopped. For example, if a line in a MIDAS program starts with "FOO:" and the colon is the 602nd character in the file, then the line describing it in the tag table would be

```
FOO:<rubout>603
```

One line can describe several tags, if they are defined on the same line; in fact, in that case, they must be on the same line in the tag table, since it must contain everything before the tag name on its definition line. For example,

```
!Foo:! !Bar:!
```

in a file of TECO code followed by character number 500 of the file would turn into

```
!Foo:! !Bar:!<rubout>500
```

EMACS will be able to use that line to find either FOO or BAR. :TAGS knows how to create such things only for TECO files, at the moment. They aren't necessary in Lisp or MACSYMA files. In MIDAS files, :TAGS simply ignores all but the first tag on a line.

21.8. Tag Tables for INFO Structured Documentation Files

INFO files are divided up into nodes, which the INFO program must search for. Tag tables for these files are designed to make the INFO program run faster. Unlike a normal tag table, the tag table for an INFO file resides in that file and describes only that file. This is so that INFO, when visiting a file, can automatically use its tag table if it has one. INFO uses the tag tables of INFO files itself, without going through the normal TAGS package, which has no knowledge of INFO file tag tables. Thus, INFO file tag tables and normal ones resemble each other only in their appearance, and that for convenience the same :TAGS program generates both. In use, they are unrelated to each other.

To create a tag table in an INFO file, you must first put in a skeleton. This skeleton must be very close to the end of the file (at most 8 lines may follow it, or INFO will not notice it), and it must start on the line following a ↑_ or ↑_↑L which ends a node. Its format is as follows:

```
↑_↑L
Tag Table:
↑_
End Tag Table
```

No nodes may follow the tag table, or :TAGS will not put them in it. :TAGS is one pass and after writing the tag table into the file it copies the rest of the input file with no processing.

To turn the skeleton into the real thing, or to update the tag table, run :TAGS on the file and specify the /I switch, as in

```
:TAGS INFO;EMACS/I
```

:TAGS will process the file and replace the old tag table or skeleton with an up-to-date tag table. The /I identifies the specified file as an INFO file rather than a tag table file. Also, it makes the default FN2 ">" rather than the usual "TAGS".

Once the tag table is constructed, INFO will automatically make use of it. A tag in an INFO file is just a node; whatever follows "Node:" on a line whose predecessor contains a "↑_" is taken to be a tag. The character which terminates the node name, which may be a comma, tab, or CRLF, is not included in the tag table. Instead, the rubout comes right after the tag name. This is to make it easy for INFO to demand an exact match on node names, rather than the substring match which the TAGS package normally uses.

Tag tables in INFO files must be kept close to up to date. INFO will not find the node if its start has moved more than 1000 characters before the position listed in the tag table. For best results, you should update an INFO file's tag table every time you modify more than a few characters of it.

22. Simple Customization

In this chapter we describe the many simple ways of customizing EMACS without knowing how to write TECO programs.

22.1. Minor Modes

Minor modes are options which you can use or not. They are all independent of each other and of the selected major mode. Most minor modes say in the mode line when they are on. See section 1.1 [Mode Line], page 6. Each minor mode is the name of the function that can be used to turn it on or off. With no argument, the function turns the mode on if it was off and off if it was on. This is known as "toggling". A positive argument always turns the mode on, and an explicit zero argument or a negative argument always turns it off. All the minor mode functions are suitable for connecting to single or double character commands if you want to enter and exit a minor mode frequently.

Auto Fill mode allows you to type text endlessly without worrying about the width or your screen. Line separators are be inserted where needed to prevent lines from becoming too long. The column at which lines are broken defaults to 70, but you can set it explicitly. C-X F (^R Set Fill Column) sets the column for breaking lines to the column point is at; or you can give it a numeric argument which is the desired column. The value is stored in the variable Fill Column.

Auto Save mode protects you against system crashes by periodically saving the file you are visiting. Whenever you visit a file, auto saving is enabled if Auto Save Default is nonzero; in addition, M-X Auto Save allows you to turn auto saving on or off in a given buffer at any time. See section 13.3 [Auto Save], page 57.

Atom Word mode causes the word-moving commands, in Lisp mode, to move over Lisp atoms instead of words. Some people like this, and others don't. In any case, the s-expression motion commands can be used to move over atoms. If you like to use segmented atom names like FOOBAR-READ-IN-NEXT-INPUT-SOURCE-TO-READ, then you might prefer not to use Atom Word mode, so that you can use M-F to move over just part of the atom, or C-M-F to move over the whole atom.

Overwrite mode causes ordinary printing characters to replace existing text instead of shoving it over. It is good for editing pictures. For example, if the point is in front of the B in FOOBAR, then in Overwrite mode typing a G changes it to FOOGAR, instead of making it FOOGBAR as usual. Also, Rubout is changed to turn the previous character into a space instead of deleting it.

Word Abbrev mode allows you to define abbreviations that automatically expand as you type them. For

example, "wam" might expand to "word abbrev mode". The abbreviations may depend on the major (e.g. Lisp, Text, ...) mode you are currently in. To use this, you must load the WORDAB library. See section 25 [Wordab], page 141.

Indent Tabs mode controls whether indentation commands use tabs and spaces or just spaces to indent with. Usually they use both, but you might want to use only spaces in a file to be processed by a program or system which doesn't ignore tabs, or for a file to be shipped to a system like Multics on which tab stops are not every 8 characters.

Most minor modes are actually controlled by variables. The mode is on if the variable is nonzero. Setting the minor mode with a command works by changing the variable. This means that you can turn the modes on or off with Edit Options, or make their values local to a buffer. See section 22.3 [Variables], page 114.

You could also put a minor mode in the local modes list of a file, but that is usually bad practice. This is because usually the preference for a minor mode is usually a matter of individual style rather that a property of the file per se. To make this more concrete, it is a property of a file that it be filled to a certain column, but use of auto fill mode to accomplish that is a matter of taste. So it would be good practice for the file to specify the value of Fill Column, but bad practice for the file to specify the value of Auto Fill Mode.

If you find yourself constantly tempted to put Auto Fill Mode in local modes lists, what you probably really want is to have Auto Fill mode on whenever you are in Text mode. This can be accomplished with the following code in an EVARS file:

```
Text Mode Hook: 1M.LAuto Fill Mode♦
```

Suffice it to explain that this is TECO code to be executed whenever Text mode is entered, which makes the variable Auto Fill Mode local to the buffer with local value 1.

22.2. Libraries of Commands

All EMACS functions, including the ones described in this document, reside in libraries. A function is not accessible unless the library that contains it is loaded. Every EMACS starts out with one library loaded: the EMACS library. This contains all of the functions described in this document, except those explicitly stated to be elsewhere. Other libraries are provided with EMACS, and can be loaded automatically or on request to make their functions available. See section [Catalogue], page 185, for a list of them.

To load a library permanently, say M-X Load Library♦<libname><cr>. The library is found, either on your own directory or whichever one you specify, or on the EMACS directory, and loaded in. All the functions in the library are then available for use. Whenever you use M-X, the function name you specify is looked up in each of the libraries which you have loaded, more recently loaded libraries first. The first definition found is

the one that is used.

For example, if you load the PICTURE library, you can then use M-X Edit Picture to run the Edit Picture function which exists in that library.

In addition to making functions accessible to M-X, the library may connect some of them to command characters.

You can also load a library temporarily, just long enough to use one of the functions in it. This avoids taking up space permanently with the library. Do this with the function Run Library, as in M-X Run♦<libname>♦<function name><cr>. The library <libname> is loaded in, and <function name> executed. Then the library is removed from the EMACS job. You can load it in again later.

M-X List Loaded Libraries types the names and brief descriptions of all the libraries loaded, last loaded first. The last one is always the EMACS library. You can get a description of all the functions in a library with M-X List Library♦<libname><cr>, whether the library is loaded or not.

The function Kill Libraries can be used to discard libraries loaded permanently by Load Library. (Libraries used with Run Library are discarded automatically). However, of all the libraries presently loaded, only the most recently loaded one can be discarded. Kill Libraries offers to kill each loaded library, most recently loaded first. It keeps killing libraries until you say to keep one library. Then it returns, because the remaining libraries cannot be deleted if that library is kept.

Libraries are loaded automatically in the course of executing certain functions. You will not normally notice this. For example, the TAGS library is automatically loaded in whenever you use M-. or Visit Tag Table for the first time. This process is known as "autoloading". It is used to make the functions in the TAGS library available without the user's having to know to load the library himself, while not taking up space in EMACSes of people who aren't using them. This works by simply calling Load Library on the library known to be needed. Another kind of "autoloading" loads a library temporarily, the way Run Library does. This is done when you use the DIRED function, for example, since the DIRED library is not needed after the DIRED function returns. This works, not by calling Run Library, but by doing M-A, which is how Run Library also works.

You can make your own libraries, which you and other people can then use, if you know how to write TECO code. See the file INFO;CONV >, node Lib, for more details.

22.3. Variables

A variable is a name which is associated with a value, either a number or a string. EMACS uses many variables internally, and has others whose purpose is to be set by the user for customization. (They may also be set automatically by major modes.) One example of such a variable is the Fill Column variable, which specifies the position of the right margin (in characters from the left margin) to be used by the fill and justify commands.

The easiest way for the beginner to set a named variable is to use the function Edit Options. This shows you a list of selected variables which you are likely to want to change, together with their values, and lets you edit them with the normal editing commands in a recursive editing level. Don't make any changes in the names, though! Just change the values. Digits with maybe a minus sign stand for a numeric value of the variable, while string values are enclosed in doublequotes. Each option is followed by a comment which says what the option is for. Type the Help character for more information on the format used.

When you are finished, exit Edit Options using C-M-C and the changes will take effect. If you decide not to make the changes, C-] gets out without redefining the options. See section 6.2 [Recursive Editing Levels], page 26.

If you give Edit Options a string argument, it shows you only the options whose names include the string. For example, M-X Edit Options♦Fill<cr> shows only the options that have "Fill" in their names. This is much more convenient, if you know what you plan to do.

However, Edit Options can be used only to set a variable which already exists, and is marked as an option. Some commands may refer to variables which do not exist in the initial environment. Such commands always use a default value if the variable does not exist. In these cases you must create the variable yourself if you wish to use it to alter the behavior of the command. You can use M-X Set Variable for this. You can set the variable to a numeric value by doing C-U <number> M-X Set Variable♦<varname><cr>, or to a string by doing M-X Set Variable♦<varname>♦<string><cr>.

In fact, you can use Set Variable to set any variable, whether it exists already or not. For existing variables, it does not matter whether you use upper case or lower case letters, and you are allowed to abbreviate the name as long as the abbreviation is unique. If the variable might not exist yet, you can't abbreviate it (how could EMACS know what it was an abbreviation of?), and while either upper case or lower case will still work, you are encouraged to capitalize each word of the name for aesthetic reasons since EMACS stores the name as you give it.

To examine the value of a single variable, the command M-X View Variable♦<varname><cr> can be used.

If you want to set a variable a particular way each time you use EMACS, you can use an init file or an EVARS file. This is one of the main ways of customizing EMACS for yourself. An init file is a file of TECO code to be executed when you start EMACS up. They are very general, but writing one is a black art. You might be able to get an expert to do it for you, or modify a copy of someone else's. See the file INFO; CONV >, node Init, for details. An EVARS file is a much simpler thing which you can do yourself. See section 22.7 [EVARS files], page 120.

Values of variables can be specified by the file being edited. For example, if a certain file ought to have a 50 column width, it can specify a value of 50 for the variable Fill Column. Then Fill Column will have the value 50 whenever this file is edited, *by anyone*. Editing other files is not affected. See section 22.6 [Locals], page 118, for how to do this.

You can get a list of all variables, not just those you are likely to want to edit, by doing M-X List Variables. Giving List Variables a string argument show only the variables whose names or values contain that string (like the function Apropos). M-X Describe can be given a variable's name instead of a function's name; it prints the variable's value and its documentation, if it has any.

You can also set a variable with the TECO command <val>U♦<varname>♦ or :1♦<varname>♦<string>♦. This is useful in init files.

Any variable can be made local to a specific buffer with the TECO command M.L<variable name>♦. Thus, if you want the comment column to be column 50 in one buffer, whereas you usually like 40, then in the one buffer do M.LComment Column♦ using the minibuffer. Then, you can do 50U♦Comment Column♦ in that buffer and other buffers will not be affected. This is how local modes lists in files work.

Most local variables are killed (made no longer local) if you change major modes. They are therefore called "mode locals". There are also "permanent" locals which are not killed by changing modes; use 2M.L to create one. Permanent locals are used by things like Auto Save mode to keep internal information about the buffer, as opposed buffer-specific customizations. See the file INFO;CONV >, node Variables, for information on how local variables work, and additional related features.

22.4. The Syntax Table

All the EMACS commands which parse words or balance parentheses are controlled by the syntax table. Each ASCII character has a word syntax and a Lisp syntax. By changing the word syntax, you can control whether a character is considered a word delimiter or part of a word. By changing the Lisp syntax, you can control which characters are parentheses, which ones are parts of symbols, which ones are prefix operators, and which ones are just ignored when parsing s-expressions.

The syntax table is actually a string which is 128*5 characters long. Each group of 5 consecutive characters of the syntax table describe one ASCII character's syntax; but only the first three of each group are used. To edit the syntax table, use M-X Edit Syntax Table. But before we describe this command, let's talk about the syntax of the syntax table itself.

The first character in each group of five sets the word syntax. This can be either "A" or a space. "A" signifies an alphabetic character, whereas a space signifies a separator character.

The second character in each group is the Lisp syntax. It has many possible values:

A	an alphabetic character
space	a whitespace or nonsignificant character
(	an open parenthesis
)	a close parenthesis
:	a comment starter
↑M	a comment ender
|	a string quote
/	a character quote
'	a prefix character

Thus, several characters can each be given the syntax of parentheses. The automatic display of matching feature uses the syntax table to decide when to go into operation as well as how to balance the parentheses.

The syntax of "prefix character" means that the character becomes part of whatever object follows it, or can also be in the middle of a symbol, but does not constitute anything by itself if surrounded by whitespace.

A character quote character causes itself and the next character to be treated as alphabetic.

A string quote is one which matches in pairs. All characters inside a pair of string quotes are treated as alphabetic except for the character quote, which retains its significance, and can be used to force a string quote or character quote into a string.

A comment starter is taken to start a comment, which ends at the next comment ender, suppressing the normal syntax of all characters between. Not all the commands which might be expected to know about comments do know about them; there a problems more than simply a need for work. Also, the syntax table entry is not what controls the commands which deal specifically with comments. They use the variables Comment Start, Comment Begin, Comment End, etc. Only the indentation commands use the syntax table for this.

The third character in each group controls automatic parenthesis matching display. It is defined only for characters which have the Lisp syntax of close parentheses, and for them it should contain the appropriate matching open parenthesis character (or a space). If a close parenthesis character is matched by the wrong kind of open parenthesis character, the bell will ring. If the third syntax table character of a close parenthesis

is a space, any open parenthesis is allowed to match it.

The fourth and fifth characters in each group should always be spaces, for now. They are not used. The reason they exist is so that word-wise indexing can be used on the PDP-10 to access the syntax of a character given in an accumulator.

Edit Syntax Table displays the syntax table broken up into labelled five-character groups. You can see easily what the syntax of any character is. You are not editing the table immediately, however. Instead, you are asked for the character whose syntax you wish to edit. After typing it, you are positioned at that character's five-character group. Overwrite mode is on, so you can simply type the desired syntax entries. You can also do arbitrary editing, but be careful not to change the position in the buffer of anything. When you exit the recursive editing level, you are asked for another character to position to. An Altmode at this point exits and makes the changes. A C-] at any time aborts the operation.

Many major modes alter the syntax table. Each major mode creates its own syntax table once and reselects the same string whenever the mode is selected, in any buffer. Thus, all buffers in Text mode at any time use the same syntax table. This is important because if you ever change the syntax table of one buffer that is in Text mode, you change them all. It is possible to give one buffer a local copy with a TECO program:

```
MM Make Local Q-Register♦..D♦W  :G..DU..D
```

The syntax tables belonging to the major modes are not preinitialized in EMACS; they are created when the major mode is invoked for the first time, by copying the default one and making specific changes. Thus, any other changes you have made in the default (Fundamental mode) syntax table at the beginning propagate into all modes' syntax tables unless those modes specifically override them.

TECO programs and init files can most easily change the syntax table with the function & Alter ..D (look at its documentation). The syntax table is kept in the q-register named ..D, which explains that name.

22.5. FS Flags

FS flags are variables defined and implemented by TECO below the level of EMACS. Some of them are options which control the behavior of parts of TECO such as the display processor. Some of them control the execution of TECO programs; you are not likely to want to change these. Others simply report information from inside TECO. The list of FS flags is fixed when TECO is assembled and each one exists for a specific purpose.

FS flags are used mostly by the TECO programmer, but some of them are of interest to the EMACS user doing minor customization. For example, FS ECHO LINES♦ is the number of lines in the echo area. By setting this flag you can make the echo area bigger or smaller.

To get the value of an FS flag, use the TECO command FS followed by the name of the flag, terminated by an Altmode. Spaces in the name of the flag are completely ignored, and case does not matter. Thus, FS Echo Lines♦= executed in the minibuffer prints the number of lines in the echo area, assuming it is a number. The easiest way to examine a flag's value with EMACS commands is

```
C-M-X View Variable<cr> (FS Echo Lines♦)<cr>
```

This works regardless of the type of value stored in the FS flag.

To set the flag, give the FS command a numeric argument (which must be a string pointer, if the intended value is a string). For example, in the minibuffer or an init file, do

```
2FS Echo Lines♦
```

Be warned that FS always returns a value, so put a CRLF after it to discard the value if necessary.

It is possible to make an FS flag's value local to a buffer. See the file INFO;CONV >, node Vars.

The documentation of individual FS flags can be found through Help T. Help T FS Echo Lines<cr> prints the description of FS ECHO LINES♦. Spaces are not significant in Help T either. A list of just the names of all FS flags is printed by the function List TECO FS Flags, found in the library PURIFY.

22.6. Local Variables in Files

By putting a "local modes list" in a file you can cause certain major or minor modes to be set, or certain character commands to be defined, whenever you are visiting it. For example, EMACS can select Lisp mode for that file, or it can turn on Auto Fill mode, set up a special Comment Column, or put a special command on the character C-M-Comma. Local modes can specify the major mode, and the values of any set of named variables and command characters. Local modes apply only while the buffer containing the file is selected; they do not extend to other files loaded into other buffers.

The simplest kind of local mode specification sets only the major mode. You put the mode's name in between a pair of "-*-"s, anywhere on the first nonblank line of the file. For example, the first line of this file contains -*-Text-*-, implying that this file should be edited in Text mode.

To specify more that just the major mode, you must use a "local modes" list, which goes in the *last* page of the file (it is best to put it on a separate page). The local modes list starts with a line containing the string "Local Modes:", and ends with a line containing the string "End:".

Each line of the local modes list should have the form <varname>:<value> (not counting the prefix and suffix, if any), which is a request to set one variable. <varname> stands for the name of the variable and <value> stands for the desired value. The name must not be abbreviated. If <value> is a numeral (which means no spaces!), the value is a number; otherwise, it is <value> as a string. To set a command character,

make <varname> the name of the character as a q-register, such as "...↑R," for C-M-Comma, and make <value> be a string of TECO commands which will return the desired value (this is so you can write M.MFoo♦ to define the character to run the function Foo).

The major mode can be set by specifying a value for the variable "Mode" (don't try setting the major mode this way except in a local modes list!). It should be the first thing in the local modes list, if it appears at all. A function M-X Foo can be defined locally by putting in a local setting for the variable named "MM Foo". See section 5.2 [Functions], page 21.

The line which starts the local modes list does not have to say only "Local Modes:". If there is other text before "Local Modes:", that text is called the "prefix", and if there is other text after, that is called the "suffix". If these are present, each entry in the local modes list should have the prefix before it and the suffix after it. This includes the "End:" line. The prefix and suffix are included to disguise the local modes list as a comment so that the compiler or text formatter will not be perplexed by it. If you do not need to disguise the local modes list as a comment in this way, do not bother with a prefix or a suffix.

Aside from the "Local Modes:" and the "End:", and the prefix and suffix if any, a local modes list looks like an EVARS file. However, comments lines are not allowed, and you cannot redefine C-X subcommands due to fundamental limitations of the data structure used to remember local variables. Sorry. See section 22.7 [EVARS files], page 120, for more information.

Here is an example of a local modes list:

```
;;; Local Modes: :::
;;; Mode:Lisp :::
;;; Comment Column:0 :::
;;; Comment Start:;;;  :::
;;; ..↑R/: m.m^R My Funny Meta-Slash♦ :::
;;; End: :::
```

Note that the prefix is ";;; " and the suffix is " :::". Note also that the value specified for the Comment Start variable is ";;; ", which is the same as the prefix, so the local modes list looks like a lot of comments. We used a suffix in this example, but they are usually not used except in languages which require comment terminators.

The last page of the file must be no more than 10000 characters long or the local modes list will not be recognized. This is because EMACS finds the local modes list by scanning back only 10000 characters from the end of the file for the last formfeed, and then looking forward for the "Local Modes:" string. This accomplishes these goals: a stray "Local Modes:" not in the last page is not noticed; and visiting a long file that is all one page and has no local mode list need not take the time to search the whole file.

22.7. Init Files and EVARS Files

EMACS is designed to be customizable; each user can rearrange things to suit his taste. Simple customizations are primarily of two types: moving functions from one character to another, and setting variables which functions refer to so as to direct their actions. Beyond this, extensions can involve redefining existing functions, or writing entirely new functions and creating sharable libraries of them.

The most general way to customize is to write an init file, a TECO program which is executed whenever you start EMACS. The init file is found by looking for a particular filename, <home directory>;<user name>EMACS. This method is general because the program can do anything. It can ask you questions and do things, rather than just setting up commands for later. However, TECO code is arcane, and only a few people learn how to write it. If you need an init file and don't feel up to learning to write TECO code, ask a local expert to do it for you. See the file INFO;CONV >, for more about init files.

However, simple customizations can be done in a simple way with an EVARS file. Such a file serves the same sort of purpose as an init file, but instead of TECO code, it contains just a list of variables and values. Each line of the EVARS file names one variable or one command character and says how to redefine it. Empty lines, and lines starting with spaces, are ignored. They can be used as comments. Your EVARS file is found by its filename, as an init file is, but it should be called <home directory>;<user name> EVARS instead of EMACS. You can have both an init file and an EVARS file if you want, as long as your init file calls the default init file, since that is what processes the EVARS file.

To set a variable, include in the EVARS file a line containing the name of the variable, a colon, and the value. If you want a string as a value, give the string; if you want a number as a value, give the digits with an optional minus sign. (If you happen to want a value which is a string of all digits, you are out of luck.) *Do not* put spaces around the colon for visual effect. Space before the colon is part of the variable name, and space after the colon is part of the value of the variable. Examples:

```
Comment Column:70
Comment Start:;
Text Mode Hook:1M.LAuto Fill Mode♦
```

Text Mode Hook, by the way, is supposed to hold a TECO program to be executed whenever Text mode is entered, and the TECO program supplied by this particular definition is designed to turn on Auto Fill mode at that time. The effect is that Auto Fill is always on when you are in Text mode.

To redefine a command character is a little more complicated. Instead of the name of a variable, give a ↑R (control-R) followed by the character. Since the general Control and Meta character cannot be part of a file, all Control and Meta characters are represented in a funny way: after the ↑R put the residue of the character after removing the Control and Meta, and before the ↑R put periods, one for Control, two for Meta, and three for Control-Meta. Thus, C-D is represented by ".↑RD" and C-M-; is represented by "...↑R;". Lower case

characters such as C-a are usually defined as "execute the definition of the upper case equivalent". Therefore, by redefining the C-A command you also change C-a; but if you redefine C-a, by saying ".↑Ra" instead of ".↑RA", you will not change C-A. So be careful about case.

Instead of the value of a variable, for command character redefinition you must give a TECO expression that returns the desired definition. This is to make it easy to use any function whose name you know, because M.MFOO♦ is an expression that returns the definition of the function FOO. Example:

```
.↑RK: M.M^R Kill Line♦
```

would give C-K the definition that it normally has. Remember that in names of functions the "^R" is actually a "^" and an R, not a Control-R. The space before the M.M does not hurt in this case because it is ignored by TECO expression execution.

Some non-printing characters are a little tricky to redefine. For example, you must know that Return, Linefeed, Tab, Backspace and Altmode are not the same in TECO's command character set as C-M, C-J, C-I, C-H and C-[, even though in ASCII they are synonymous. By saying .↑RJ you will redefine C-J; by saying ↑R followed by a Linefeed (which you must insert in the EVARS file by typing C-Q Linefeed) you can redefine Linefeed. Normally, C-J is defined as "execute the definition of Linefeed", so you are better off redefining Linefeed.

You can also redefine a subcommand of a prefix character such as C-X. For this, you have to know where the character's dispatch table is stored. For C-X, the location of the dispatch is called ".X"; you won't have any other prefix characters unless you define them yourself. See the file INFO;CONV >, node Prefix. Knowing the location, you specify the subcommand by writing :location(↑^character). This looks silly, but it is a TECO expression with the right meaning. For example, redefining C-X C-S, the location is ".X" and the character is ↑S, so we say

```
:.X(↑^↑S): M.M^R Save File♦
```

This gives C-X C-S the definition that it normally has. The subcommand character (↑S in this case) can represent itself in the EVARS file with no need for dots, because subcommand characters are just ASCII, with no Meta allowed.

To connect a command character to a function from a library which is not normally loaded, you can do

```
.↑R,: MM Load♦FOO♦W M.MBar♦
```

This loads the library FOO and connects the command C-Comma to the function Bar, presumably found in that library. The "W" discards the value returned by MM Load♦ so that it does not interfere with the M.MBar♦.

To simply load a library you can write a definition for "*". Such a definition is ignored except that the value you specify is executed as a TECO expression. Thus, an arbitrary TECO expression can be snuck into an EVARS file. To load the library FOO, use the expression MM Load♦FOO♦.

```
*: MM Load♦FOO♦
```

Please refrain from giving newcomers to EMACS a copy of your own init file before they understand what it does. Everyone prefers his own customizations, and there is always a tendency to proselytize, but by the same token your protege's tastes may be different from yours. If you offer him your customizations at the time when he is ready to understand what difference they make and decide for himself what he prefers, then you will help him get what *he* wants. Tell him about each individual change you made, and let him judge them one by one. There is no reason for him to choose all or nothing.

22.7.1. EVARS File Examples

Here are some examples of how to do various useful things in an EVARS file.

This causes new buffers to be created in Lisp mode:

```
Default Major Mode:LISP
```

This causes new buffers to have auto fill mode turned on:

```
Buffer Creation Hook: 1M.L Auto Fill Mode♦
```

This causes all Text mode buffers to have auto fill mode turned on:

```
Text Mode Hook: 1M.L Auto Fill Mode♦
```

This causes C-M-G to be undefined by copying the definition of C-M-Space (which is undefined):

```
...↑RG: Q...↑R          (a space follows the control-R)
```

This redefines C-S to be a single character search command, and M-S to be a non-incremental string search:

```
.↑RS: M.M ^R Character Search♦
..↑RS: M.M ^R String Search♦
```

This redefines C-X V to run View File:

```
:.X(↑^V): M.M View File♦
```

This makes M-M a prefix character and defines M-M W to mark a word and M-W P to mark a paragraph. It stores the dispatch vector for the prefix character in q-register .Y.

```
..↑RM: MM Make Prefix Character♦.Y♦
:.Y(↑^W): M.M ^R Mark Word♦
:.Y(↑^P): M.M ^R Mark Paragraph♦
```

This loads the library LUNAR and defines C-Q to run a useful function in that library:

```
*: MM Load Library♦LUNAR♦
.↑RQ: M.M ^R Various Quantities♦
```

This causes Auto Save mode to save under the visited filenames:

```
Auto Save Visited File:1
```

This causes TAGS to bring new files into separate buffers:

```
TAGS Find File:1
```

This stops the message "EMACS version nnn. Type ... for Help" from being printed.

```
Inhibit Help Message:1
```

This redefines the list syntax of "%" to be ";" for "comment starter", and that of ";" to be "A" for "alphabetic":

```
*: 1mm& Alter ..D♦%;;A♦
```

22.7.2. Init File Examples

Here are the ways to do exactly the same things in an init file.

This causes new buffers to be created in Lisp mode:

```
:I♦Default Major Mode♦LISP♦
```

This causes new buffers to have auto fill mode turned on:

```
:I* 1M.L Auto Fill Mode↑]♦ ♦ M.VBuffer Creation Hook♦
```

It is different because the variable does not already exist. Note the ↑] used for getting the Altmode into the value.

This causes all Text mode buffers to have auto fill mode turned on:

```
:I* 1M.L Auto Fill Mode↑]♦ ♦ M.VText Mode Hook♦
```

This causes C-M-G to be undefined by copying the definition of C-M-Space (which is undefined):

```
Q...↑R  U...↑RG
```

This redefines C-S to be a single character search command, and M-S to be a non-incremental string search:

```
M.M ^R Character Search♦ U.↑RS
M.M ^R String Search♦ U..↑RS
```

This redefines C-X V to run View File:

```
M.M View File♦ U:.X(↑^V)
```

This makes M-M a prefix character and defines M-M W to mark a word and M-W P to mark a paragraph. It stores the dispatch vector for the prefix character in q-register .Y.

```
MM Make Prefix Character♦.Y♦U..↑RM
M.M ^R Mark Word♦ U:.Y(↑^W)
M.M ^R Mark Paragraph♦ U:.Y(↑^P)
```

This loads the library LUNAR and defines C-Q to run a useful function in that library:

```
MM Load Library♦LUNAR♦
M.M ^R Various Quantities♦ U.↑RQ
```

This causes Auto Save mode to save under the visited filenames:

```
1U♦Auto Save Visited File♦
```

Compare this and the next example with the first two, in which string values are used.

This causes TAGS to bring new files into separate buffers:

```
1M.VTAGS Find File♦
```

This stops the message "EMACS version nnn. Type ... for Help" from being printed.

```
1M.VInhibit Help Message♦
```

This redefines the list syntax of "%" to be ";" for "comment starter", and that of ";" to be "A" for "alphabetic":

```
1mm& Alter ..D♦%;;A♦
```

22.8. Keyboard Macros

C-X (	Start defining a keyboard macro.
C-X)	End the definition of a keyboard macro.
C-X E	Execute the most recent keyboard macro.
C-X Q	Ask for confirmation when the keyboard macro is executed.
C-U C-X Q	Allow the user to edit for a while, each time the keyboard macro is executed.
M-X Name Kbd Macro	

Make the most recent keyboard macro into the permanent definition of a command.

A keyboard macro is a command defined by the user to abbreviate a sequence of other commands. If you discover that you are about to type C-N C-D forty times, you can define a keyboard macro to do C-N C-D and call it with a repeat count of forty.

Keyboard macros differ from ordinary EMACS commands, in that they are written in the EMACS command language rather than in TECO. This makes it easier for the novice to write them, and makes them more convenient as temporary hacks. However, the EMACS command language is not powerful enough as a programming language to be useful for writing anything intelligent or general. For such things, TECO must be used.

EMACS functions were formerly known as macros (which is part of the explanation of the name EMACS), because they were macros within the context of TECO as an editor. We decided to change the terminology because, when thinking of EMACS, we consider TECO a programming language rather than an editor. The only "macros" in EMACS now are keyboard macros.

You define a keyboard macro while executing the commands which are the definition. Put differently, as you are defining a keyboard macro, the definition is being executed for the first time. This way, you can see what the effects of your commands are, so that you don't have to figure them out in your head. When you are finished, the keyboard macro is defined and also has been, in effect, executed once. You can then do the whole thing over again by invoking the macro.

22.8.1. Basic Use

To start defining a keyboard macro, type the C-X (command (^R Start Kbd Macro). From then on, your commands continue to be executed, but also become part of the definition of the macro. "Def" appears in the mode line to remind you of what is going on. When you are finished, the C-X) command (^R End Kbd Macro) terminates the definition (without becoming part of it!). The macro thus defined can be invoked again with the C-X E command (^R Execute Kbd Macro), which may be given a repeat count as a numeric argument to execute the macro many times. C-X) can also be given a repeat count as an argument, in which case it repeats the macro that many times, but defining the macro counts as the first repetition (since it is executed as you define it). So, giving C-X) an argument of 2 executes the macro one additional time. An argument of zero to C-X E or C-X) means repeat the macro indefinitely (until it gets an error).

If you wish to save a keyboard macro for longer than until you define the next one, you must give it a name. Do M-X Name Kbd Macro♦FOO<cr> and the last keyboard macro defined (the one which C-X E would invoke) is turned into a function and given the name FOO. M-X FOO will from then on invoke that particular macro. Name Kbd Macro also reads a character from the keyboard and redefine that character

command to invoke the macro. If you don't want to redefine a command, type a Return or Rubout. Only self-inserting and undefined characters, and those that are already keyboard macros, can be redefined in this way. Prefix characters may be used in specifying the command to be redefined.

To examine the definition of a keyboard macro, use the function View Kbd Macro. Either supply the name of the function which runs the macro, as a string argument, or type the command which invokes the macro, on the terminal when View Kbd Macro asks for it.

22.8.2. Executing Macros with Variations

If you want to be allowed to do arbitrary editing at a certain point each time around the macro (different each time, and not remembered as part of the macro), you can use the C-U C-X Q command (^R Kbd Macro Query). When you are defining the macro, this lets you do some editing, which does *not* become part of the macro. When you are done, exit with C-M-C to return to defining the macro. When you execute the macro, at that same point, you will again be allowed to do some editing. When you exit this time with C-M-C, the execution of the macro will resume. If you abort the recursive editing level with C-], you will abort the macro definition or execution.

You can get the effect of Query Replace, where the macro asks you each time around whether to make a change, by using the command C-X Q with no argument in your keyboard macro. When you are defining the macro, the C-X Q does nothing, but when the macro is invoked the C-X Q reads a character from the terminal to decide whether to continue. The special answers are Space, Rubout, Altmode, C-L, C-R. A Space means to continue. A Rubout means to skip the remainder of this repetition of the macro, starting again from the beginning in the next repetition. An Altmode ends all repetitions of the macro, but only the innermost macro (in case it was called from another macro). C-L clears the screen and asks you again for a character to say what to do. C-R enters a recursive editing level; when you exit, you are asked again (if you type a Space, the macro will continue from wherever you left things when you exited the C-R). Anything else exits all levels of keyboard macros and is reread as a command.

23. The Minibuffer

M-Altmode	Invokes an empty minibuffer.
M-%	Invokes a minibuffer initialized with a Query Replace.
C-X Altmode	Re-execute a recent minibuffer command.
C-X ^	Add more lines to the minibuffer.
C-\	Meta-prefix for use in the minibuffer.
C-C C-Y	Rotate ring of recent minibuffer commands.

The minibuffer is a facility by means of which EMACS commands can read input from the terminal, allowing you to use EMACS commands to edit the input while you are typing it. The primary use of the minibuffer is for editing and executing simple TECO programs such as

```
MM Query Replace♦FOO
♦BAR
♦
```

(which could not be done with M-X because Returns are part of the arguments).

You can always tell when you are in a minibuffer, because the mode line contains something in parentheses, such as "(Minibuffer)" or "(Query Replace)". There is also a line of dashes across the screen a few lines from the top. Strictly speaking, the minibuffer is actually the region of screen above the line of dashes, for that is where you edit the input that the minibuffer is asking you for. Editing has been limited to a few lines so that most of the screen can continue to show the file you are visiting.

If you want to type in a TECO command, use the minibuffer with the command Meta-Altmode, (^R Execute Minibuffer). An empty minibuffer will appear, into which you should type the TECO command string. Exit with Altmode Altmode, and remember that neither of the two Altmodes is inserted into your TECO command although the first one may appear to be. When the TECO command is executed, "the buffer" will be the text you were editing before you invoked the minibuffer.

Often, a minibuffer starts out with some text in it. This means that you are supposed to add to that text, or, sometimes, to delete some of it so as to choose among several alternatives. For example, Meta-% (^R Query Replace) provides you with a minibuffer initially containing the string "MM Query Replace♦". The cursor comes at the end. You are then supposed to add in the arguments to the Query Replace.

In a minibuffer, you can edit your input until you are satisfied with it. Then you tell EMACS you are finished by typing two Altmodes. An Altmode not followed by another Altmode is simply inserted in the buffer. This is because it is common to want to put Altmodes into the minibuffer, which usually contains a string of TECO commands. For example, in Meta-% (^R Query Replace) each argument must be ended by an Altmode. However, when you type two Altmodes in a row, neither one remains in the buffer. The two Altmodes do nothing to the text in the minibuffer, they just exit.

Since Altmode is self-inserting, typing Meta characters can be a problem. You can do it by using C-\ instead of Altmode as the Meta-prefix. If you type a Control-Meta character on your keyboard, the corresponding ASCII control character is inserted in the minibuffer. This is because the Lisp commands are rarely useful when editing TECO code, but insertion of control characters is frequent. If you really want to use a Control-Meta EMACS command, you must use C-C to type it. You cannot use C-\ C-A to type C-M-A, because C-\ (unlike Altmode) ignores the Control bit of the following character, so you must use C-C C-A. The motivation for this quirk of C-\ is that C-\ C-B (to obtain M-B) is easier to type than C-\ B, especially if it is typed several times in a row.

You can cancel your input in a minibuffer and start all over again by typing C-G. That kills all the text in the minibuffer. A C-G typed when the minibuffer is already empty exits from the minibuffer. Usually, this aborts whatever command was using the minibuffer, so it will return without doing anything more. For example, if you type two C-G's at Meta-%'s minibuffer, you will return to top level and no Query Replace will be done. Typing a single C-G at a preinitialized minibuffer to empty the buffer is not very useful, since you would have to retype all the initial text.

The last five distinct minibuffer commands or M-X commands you have issued are remembered in a ring buffer in q-register .M. The C-X Altmode command (^R Re-execute Minibuffer) re-executes the last command in the ring. With an argument <n>, it re-executes the <n>'th previous command. The command is printed out (only the first 40 characters or so) and you are asked to confirm with "Y" or "N".

You can also get your previous minibuffer and M-X commands back into the minibuffer to be edited and re-executed with changes. Giving M-Altmode and argument, as in C-U M-Altmode, causes the minibuffer to be loaded up with the last command in the ring, as if you had typed it in again from scratch. You can then edit it, execute it by typing two Altmodes, or cancel it with C-G. To get an earlier command string instead of the most recent one, use the command C-C C-Y once you are in the minibuffer. This command "rotates" the ring of saved commands much as M-Y rotates the ring of killed text. Each C-C C-Y reveals an earlier command string, until the ring has rotated all the way around and the most recent one reappears. C-C C-Y is actually a way of saying C-M-Y, but in the minibuffer that's the only way to type it, since Altmode inserts itself and Control-Meta characters insert control characters.

If you exit from Meta-Altmode with a C-G, nothing is executed and the previous minibuffered command string is still remembered as the last one.

While in a minibuffer, if you decide you want the minibuffer to use more lines on the screen, you can use C-X ^ (^R Grow Window) to get more. It gets one more line, or as many lines as its argument says.

24. Correcting Mistakes and EMACS Problems

If you type an EMACS command you did not intend, the results are often mysterious. This chapter tells what you can do to cancel your mistake or recover from a mysterious situation. EMACS bugs and system crashes are also considered.

24.1. Cancelling a Command

C-G Quit. Cancel running or partially typed command.
C-] Abort recursive editing level and cancel the command which invoked it.
M-X Top Level
Abort all recursive editing levels and subsystems which are currently executing.

There are three ways of cancelling commands which are not finished executing: "quitting" with C-G, and "aborting" with C-] or M-X Top Level. Quitting is cancelling a partially typed command or one which is already running. Aborting is cancelling a command which has entered a recursive editing level.

Quitting with C-G is used for getting rid of a partially typed command, or a numeric argument that you don't want. It also stops a running command in the middle in a relatively safe way, so you can use it if you accidentally give a command which takes a long time. In particular, it is safe to quit out of killing; either your text will *all* still be there, or it will *all* be in the kill ring (or maybe both). Quitting an incremental search does special things documented under searching; in general, it may take two successive C-G's to get out of a search. C-G can interrupt EMACS at any time, so it is not an ordinary command.

Aborting with C-] (Abort Recursive Edit) is used to get out of a recursive editing level and cancel the command which invoked it. Quitting with C-G cannot be used for this, because it is used to cancel a partially typed command within the recursive editing level. Both functions are useful. For example, if you are editing a message to be sent, C-G can be used to cancel the commands you use to edit the message, and C-] cancels sending the message. C-] either tells you how to resume the aborted command or queries for confirmation before aborting.

When you are in a position to use M-X, you can use M-X Top Level. This is equivalent to "enough" C-] commands to get you out of all the levels of subsystems and recursive edits that you are in. C-] gets you out one level at a time, but M-X Top Level goes out all levels at once. Both C-] and M-X Top Level are like all other commands, and unlike C-G, in that they are effective only when EMACS is listening.

24.2. What to Do if EMACS Acts Strangely

This section describes various conditions which can cause EMACS not to work, or cause it to display strange things, and how you can correct them.

24.2.1. Error Message

When EMACS prints an error message, it occupies the top line of the screen, ends with a "?", and is accompanied by the ringing of the bell. Space causes the error message to disappear and be replaced by the first line of text again. Any other command is executed normally as if there had been no error message (the error message disappears during the redisplay after the command). However, "?" enters the error handler, which can be used to inspect the function call stack. Type Help inside the error handler to get its documentation. Most users will not be interested in doing this.

24.2.2. Subsystems and Recursive Editing Levels

Subsystems and recursive editing levels are important and useful aspects of EMACS, but they can seem like malfunctions to the user who does not understand them.

If the mode line starts with a bracket "[" or a parenthesis "(", or does not start with the word "EMACS", then you have entered a subsystem (See section 6.1 [Subsystems], page 25.) or a recursive editing level (See section 6.2 [Recursive Editing Levels], page 26.).

In such a situation, first try typing C-]. This will get out of any recursive editing level and most subsystems. The usual mode line and text display will reappear. If C-] does not seem to have worked, type the Help character. Instead of printing "Doc (Type ? for Help)" in the echo area, it will print a list of the subsystem's commands. One of these should be a command to exit or abort.

If the above techniques fail, try restarting (see section 24.2.7).

24.2.3. Garbage on the Screen

If the data on the screen looks wrong, it could be due to line noise on input or output, a bug in the terminal, a bug in EMACS redisplay, or a bug in an EMACS command. To find out whether there is really anything wrong with your text, the first thing to do is type C-L. This is a command to clear the screen and redisplay it. Often this will display data which is more pleasing. Think of it as getting an opinion from another doctor.

24.2.4. Garbage Displayed Persistently

If EMACS persistently displays garbage on the screen, or if it outputs the right things but scattered around all the wrong places on the screen, it may be that EMACS has the wrong idea of your terminal type. The first thing to do in this case is to exit from EMACS and restart it. Each time EMACS is restarted it asks the system what terminal type you are using. Whenever you detach and move to a terminal of a different type you should restart EMACS as a matter of course. If you stopped EMACS with the exit command, or by interrupting it when it was awaiting a command, then this is sure to be safe.

The system itself may not know what type of terminal you have. You should try telling the system with the :TCTYP command.

24.2.5. URK Error (Address Space Exhausted)

If attempting to visit a file or load a library causes an "URK" error, it means you have filled up the address space; there is no room inside EMACS for any more files or libraries. In this situation you can run M-X Make Space. This command compacts the data inside EMACS to free up some space. It also offers to discard data that may be occupying a lot of space, such as the kill ring (See section 9.1 [Killing], page 35.), the undo memory (See section 24.3 [Undo], page 132.), and buffers created by RMAIL, TAGS and INFO. Another way of freeing space is to kill buffers with M-X Kill Some Buffers (See section 14 [Buffers], page 67.) or unload libraries with M-X Kill Libraries (See section 22.2 [Libraries], page 112.).

24.2.6. All Type-in Echoes and Nothing Else Happens

If you find that EMACS is not responding to your commands except for echoing them all at the bottom of the screen, including the Return character, and that Rubout causes erased characters to be retyped instead of erased, then you have managed to exit from EMACS back to TECO. Often this follows an "Error in error handler" message which indicates that a condition arose in which the error handler could not function. You can get back into EMACS by typing :M..I.↑↑, or by restarting (see below). If you ever *want* to exit back to TECO, you can do M-X Top Level with an argument greater than zero. Before using :M..I.↑↑, get rid of any other characters you have typed by mistake by typing a C-G.

24.2.7. EMACS Hung and Not Responding

Sometimes EMACS gets hung and C-G does not work. The more drastic procedure of restarting EMACS may work at such times. C-G can fail to work because it only takes effect between the TECO commands

which make up an EMACS program, never in the middle of one (only a few TECO commands allow quitting at any time), so as to prevent TECO's internal data structures from becoming inconsistent. If EMACS is hung inside a TECO command, C-G is not noticed, but restarting can still be tried.

To restart EMACS, type CALL or C-Z to stop EMACS, then ↑G to restart it. While restarting TECO in this way is usually safe (especially at times when TECO is doing I/O), there are certain times at which it will cause the TECO data structures to be inconsistent, so do not try it unless other measures have failed.

Your ultimate safeguard against a wedged EMACS is to save your work frequently.

24.3. Undoing Changes to the Buffer

If you mistakenly issue commands that make a great change to the buffer, you can often undo the change without having to know precisely how it came about. This is done by using M-X Undo. Type M-X Undo<cr> and the change is undone. It does not matter if you have moved the cursor since you made the change; it is undone where it was originally done.

The first thing Undo does is tell you what kind of change it plans to undo (kill, fill, undo, case-convert, etc). Then it asks whether to go ahead. If you say "Y", the change is actually undone.

Not all changes to the buffer can be undone: deletion (as opposed to killing) can't be, and changes in indentation can't be, nor can many forms of insertion (but they aren't as important since they don't destroy information). Also, a Replace String or Query Replace can't be undone, which is a shame. The reason is that actually they make many small changes, and Undo only knows how to remember one contiguous change. Perhaps someday I will be able to fix this.

As a result, when you say Undo, it may undo something other than the latest change if the latest change was not undoable. This might seem to pile one disaster on another, but it doesn't, because you can *always* Undo the Undo if it didn't help. But you can avoid even having to do that, if you look at what type of change Undo says it will undo.

If you want to undo a considerable amount of editing, not just the last change, the Undo command can't help you, but M-X Revert File (See section 13.2 [Revert], page 57.) might be able to. If you have been writing a journal file (See section 24.4 [Journals], page 133.), you can replay the journal after deleting the part that you don't want.

24.4. Journal Files

A journal file is a record of all the commands you type during an editing session. If you lose editing because of a system crash, an EMACS bug, or a mistake on your part, and you have made a journal file, you can replay the journal or part of it to recover what you lost. Journal files offer an alternative to auto saving, using less time and disk space if there is no crash, but requiring more time when you recover from a crash. See section 13.3 [Auto Save], page 57.

24.4.1. Writing Journal Files

In order to make a journal file, you must load the JOURNAL library and then execute M-X Start Journal File♦<filename><cr>. Immediately, most of the current status of EMACS is recorded in the journal file, and all subsequent commands are recorded as they are typed. This happens invisibly and silently. The journal file is made fully up to date on the disk after every 50th character, so the last 50 characters of type in is the most you can lose.

The default filenames for the journal file are <home directory>;<user name> JRNL. There is rarely a reason to use any other name, because you only need one journal file unless you are running two EMACSes at the same time.

24.4.2. Replaying Journal Files

To replay the journal file, get a fresh EMACS, load JOURNAL, and do M-X Replay Journal File♦<filename><cr>. The filename can usually be omitted since normally you will have used the defaults when creating the journal.

After a delay while the files, buffers and libraries are loaded as they were when the journal file was written, EMACS will begin replaying the commands in the journal before your very eyes. Unlike keyboard macros, which execute invisibly until they are finished, journal files display as they are executed. This allows you to see how far the replay has gone. You can stop the process at any time by typing C-G. Aside from that, you should not type anything on the keyboard while the replay is going on.

If the need for a replay is the result of a system crash or EMACS crash, then you probably want to replay the whole file. This is what happens naturally. If you are replaying because you made a great mistake, you probably want to stop the replay before the mistake. This is when it becomes useful to type C-G to stop the replay. Alternatively, you can edit the journal file, and delete everything from the point of the mistake to the end, before you replay it.

Once the replay is complete, save all your files immediately. Don't tempt fate!

If you quit with C-G in the middle of a command while writing a journal file, there is no way to record in the journal file how much of the command has already been completed. So, when the journal is replayed, EMACS has to ask you to fill in for it. The command which was interrupted will be replayed to completion; then, you are given a recursive editing level in which to restore the file to the desired state. This happens only if the C-G originally interrupted an executing command. C-G typed to discard an argument or partial command while EMACS is waiting for input can be and is replayed correctly without asking you for help.

24.4.3. Journal File Format

To edit a journal file, you must know the format. It is designed to be mostly transparent.

The primary problem which the journal file format has to solve is how to represent 9-bit command characters in a file which can contain only 7-bit ASCII characters. (We could have filled the journal file with 9-bit characters, but then you would not be able to print it out or edit it). The solution we have used is to represent each command by two characters in the file.

So, a Control character is represented by a caret ("^") followed by the basic character, as in "^E" for Control-E. This was chosen to be mnemonically significant. A Meta character is represented by "+" followed by the basic character, so that Meta-[is represented by "+[". A Control-Meta character is represented by "*" followed by the basic character, as in "*X" for C-M-X.

A command which is not Control or Meta is represented as a space followed by the command itself, except that Return is represented by a CRLF rather than a space and a carriage return. This prevents the journal file from being one huge line, and makes insertion of text very recognizable: the text inserted appears in the journal file alternating with spaces.

The Help character, which is not covered by the scheme as described so far, is represented by "??".

An asynchronous quit, which is a problem for replaying, is represented by a single character, a ↑G, while a synchronous quit, which can be replayed reliably, is represented by ":↑G". EMACS considers a quit synchronous, and uses ":↑G" to record it, if EMACS was waiting for terminal input when the C-G was typed.

Your commands themselves are not the only information in the journal file. EMACS records other information which is necessary in replaying the journal properly. The colon character ":" indicates a block of such information. Usually the extent of the block is easily recognizable because its contents do not resemble the representations of commands described above. A large block of information starting with a colon appears at the beginning of every journal file.

Colons are also used to record the precise effects of certain commands such as C-V whose actions depend on how the text was displayed on the screen. Since the effects of such commands are not completely determined by the text, replaying the command could produce different results, especially if done on a terminal with a different screen size. The extra information recorded in the journal makes it possible to replay these commands with fidelity.

A semicolon in the journal file begins a comment, placed there for the benefit of a human looking at the journal. The comment ends at the beginning of the following line.

24.4.4. Warnings

Proper replaying of a journal file requires that all the surrounding circumstances be unchanged.

In particular, replaying begins by visiting all the files that were visited when the writing of the journal file began; not the latest versions of these files, but the versions which were the latest at the earlier time. If those versions, which may no longer be the latest, have been deleted, then replaying is impossible.

If your init file has been changed, the commands when replayed may not do what they did before.

These are the only things that can interfere with replaying, as long as you start writing the journal file immediately after starting EMACS. But as an editing session becomes longer and files are saved, the journal file contains increasing amounts of waste in the form of commands whose effects are already safe in the newer versions of the edited files. Replaying the journal will replay all these commands wastefully to generate files identical to those already saved, before coming to the last part of the session which provides the reason for replaying. Therefore it becomes very desirable to start a new journal file. However, many more precautions must be taken to insure proper replaying of a journal file which is started after EMACS has been used for a while. These precautions are described here. If you cannot follow them, you must make a journal checkpoint (see below).

If any buffer contains text which is not saved in a file at the time the journal file is started, it is impossible to replay the journal correctly. This problem cannot possibly be overcome. To avoid it, M-X Start Journal File offers to save all buffers before actually starting the journal.

Another problem comes from the kill ring and the other ways in which EMACS remembers information from previous commands. If any such information which originated before starting the journal file is used after starting it, the journal file cannot be replayed. For example, suppose you fill a paragraph, start a journal file, and then do M-X Undo? When the journal is replayed, it will start by doing M-X Undo, but it won't know what to undo. It is up to you not to do anything that would cause such a problem. It should not be hard. It would be possible to eliminate this problem by clearing out all such data structures when a journal

file is started, if users would prefer that.

A more difficult problem comes from customization. If you change an option or redefine a command, then start a journal file, the journal file will have no record of the change. It will not replay correctly unless you remember to make the same change beforehand. Customizations made in an init file do not cause a problem because the init file has also been run when the journal file is replayed. Customizations made directly by the user while the journal file is being written are also no problem because replaying will make the same changes at the right times. However, a customization made while a journal file is being written *will* be a problem if a new journal file is started.

24.4.5. Journal Checkpoints

The only cure for the problems of starting a journal in mid-session is to record the complete state of EMACS at the the time the journal is begun. This is not done normally because it is slow; however, you can do this if you wish by giving M-X Start Journal File a numeric argument. This writes the complete state of EMACS into the file <home directory>;ITS ESAVE. To replay the journal, run ESAVE, the saved checkpoint, instead of EMACS; then load JOURNAL and do M-X Replay Journal File as described above. Be sure to delete the checkpoint if you are finished with it, since it tends to be large. Delete them also when you log out; it may be possible to have a command file which deletes them automatically when you log out. Checkpoint files more than a day old will probably be deleted by others without notice.

24.5. Reporting Bugs

There will be times when you encounter a bug in EMACS. To get it fixed, you must report it. It is your duty to do so; but you must know when to do so and how if it is to be constructive.

24.5.1. When Is There a Bug

If EMACS executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to "disk full"), then it is certainly a bug.

If EMACS updates the display in a way that does not correspond to what is in the buffer, then it is certainly a bug. If a command seems to do the wrong thing but the problem is gone if you type C-L, then it is a case of incorrect display updating.

Taking forever to complete a command can be a bug, but you must make certain that it was really EMACS's fault. Some commands simply take a long time. Quit or restart EMACS and type Help L to see

whether the keyboard or line noise garbled the input; if the input was such that you *know* it should have been processed quickly, report a bug. If you don't know, try to find someone who does know.

If a command you are familiar with causes an EMACS error message in a case where its usual definition ought to be reasonable, it is probably a bug.

If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. Rather than jumping to conclusions, show the problem to someone who knows for certain.

Finally, a command's intended definition may not be best for editing with. This is a very important sort of problem, but it is also a matter of judgement. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways (INFO and Help), feel confident that you understand it, and know for certain that what you want is not available. If you feel confused about the documentation instead, then you don't have grounds for an opinion about whether the command's definition is optimal. Make sure you read it through and check the index or the menus for all references to subjects you don't fully understand. If you have done this diligently and are still confused, or if you finally understand but think you could have said it better, then you have a constructive complaint to make *about the documentation*. It is just as important to report documentation bugs as program bugs.

24.5.2. How to Report a Bug

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, starting with a fresh EMACS just loaded, until the problem happens.

The most important principle in reporting a bug is to report *facts*, not hypotheses or conditions. It is always easier to report the facts, but people seem to prefer to strain to think up explanations and report them instead. If the explanations are based on guesses about how EMACS is implemented, they will be useless; we will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for us.

For example, suppose that you type C-X C-V GLORP;BAZ UGH<cr>, visiting a file which (you know) happens to be rather large, and EMACS prints out "I feel pretty today". The best way to report the bug is with a sentence like the preceding one, because it gives all the facts and nothing but the facts.

Do not assume that the problem is due to the size of the file and say "When I visit a large file, EMACS

prints out 'I feel pretty today'". This is what we mean by "guessing explanations". The problem is just as likely to be due to the fact that there is a "Z" in the filename. If this is so, then when we got your report, we would try out the problem with some "big file", probably with no "Z" in its name, and not find anything wrong. There is no way in the world that we could guess that we should try visiting a file with a "Z" in its name.

Alternatively, the problem might be due to the fact that the file starts with exactly 25 spaces. For this reason, you should make sure that you don't change the file until we have looked at it. Suppose the problem only occurs when you have typed the C-X C-A command previously? This is why we ask you to give the exact sequence of characters you typed since loading the EMACS.

You should not even say "visit the file ..." instead of "C-X C-V" unless you *know* that it makes no difference which visiting command is used. Similarly, rather than saying "if I have three characters on the line", say "after I type <cr> A B C <cr> C-P", if that is the way you entered the text. A journal file containing the commands you typed to reproduce the bug is a very good form of report.

Send the bug report to BUG-EMACS@MIT-AI if you are on the Arpanet or to the author (see the preface for the address).

If you are not in Fundamental mode when the problem occurs you should say what mode you are in.

Be sure to say what version of EMACS and TECO are running. If you don't know, type Meta-Altmode Q◆EMACS Version◆ = FS Version◆ = ◆◆ and EMACS will print them out. (This is a use of the minibuffer. See section 23 [Minibuffer], page 127.)

If the bug occurred in a customized EMACS, or with several optional libraries loaded, it is helpful to try to reproduce the bug in a more standard EMACS with fewer libraries loaded. It is best if you can make the problem happen in a completely standard EMACS with no optional libraries. If the problem does *not* occur in a standard EMACS, it is very important to report that fact, because otherwise we will try to debug it in a standard EMACS, not find the problem, and give up. If the problem does depend on an init file, then you should make sure it is not a bug in the init file by complaining to the person who wrote the file, first. He should check over his code, and verify the definitions of the TECO commands he is using by looking in INFO;TECORD >. Then if he verifies that the bug is in EMACS he should report it. We cannot be responsible for maintaining users' init files; we might not even be able to tell what they are supposed to do.

If you can tell us a way to cause the problem without reading in any files, please do so. This makes it much easier to debug. If you do need files, make sure you arrange for us to see their exact contents. For example, it can often matter whether there are spaces at the ends of lines, or a line separator after the last line in the buffer (nothing ought to care whether the last line is terminated, but tell that to the bugs).

If EMACS gets an operating system error message, such as for an illegal instruction, then you can probably recover by restarting it. But before doing so, you should make a dump file. Do this by saying :PDUMP CRASH;EMACS >. Be sure to report exactly all the numbers printed out with the error message. Also type .JPC/ and include DDT's response in your report. If you restart or continue the EMACS before saving this information, the trail will be covered and it will probably be too late to find out what happened.

A dump is also useful if EMACS gets into a wedged state in which commands that usually work do strange things.

25. Word Abbreviation Input

Word Abbrev mode allows the EMACS user to abbreviate text with a single "word", with EMACS expanding the abbreviation automatically as soon as you have finished the abbreviation, with control over capitalization of the expanded string.

Abbrevs are also useful for correcting commonly misspelled or mistyped words (e.g. "thier" could expand to "their"), and for uppercasing words like "EMACS" (abbrev "emacs" could expand to "EMACS").

To use this mode, just do M-X Word Abbrev Mode<cr>. (Another M-X Word Abbrev Mode<cr> will turn the mode off; it toggles.)

For example, in writing this documentation I could have defined "wam" to be an abbreviation for "word abbrev mode". After only typing the letters "wam", I see just that, "wam", but if I then finish the word by typing space or period or any of the text break-characters, the "wam" is replaced by (and redisplays as) "word abbrev mode". If I capitalize the abbrev, "Wam", the expansion is capitalized: "Word abbrev mode". If I capitalize the whole abbrev, WAM", each word in the expansion is capitalized: "Word Abbrev Mode". In this particular example, though I would define "wam" to expand to "Word Abbrev mode" since it is always to be capitalized that way.

Thus, I can type "I am in wam now" and end up with "I am in Word Abbrev mode now".

Word Abbrev mode does not interfere with the use of major modes, e.g. Text, Lisp, TECO, PL1, or minor modes, e.g. Auto Fill. Those modes (or the user) may redefine what functions are to be called by characters; that does not interfere with Word Abbrev mode.

There are two kinds of word abbreviations: mode and global. A mode word abbrev causes expansion in only one major mode (for instance only in Text mode), while a global word abbrev causes expansion in all major modes. If some abbrev is both a mode word abbrev for the current mode and a global word abbrev, the mode word abbrev expansion takes precedence.

For instance, you might want an abbrev "foo" for "find outer otter" in Text mode, an abbrev "foo" for "FINAGLE-OPPOSING-OPINIONS" in Lisp, and an abbrev "foo" for "meta-syntactic variable" in any other mode (the global word abbrev).

Word abbrevs can be defined one at a time (adding them as you think of them), or many at a time (from a definition list). You can save them in a file and read them back later. Word abbrev definitions stay around even while you're not in Word Abbrev mode, though they don't expand.

Word abbrevs can be killed either singly, or by editing the current definition list.

25.1. Basic Usage

C-X C-A	Define a mode abbrev for some text before point.
C-X +	Define a global abbrev for some text before point.
C-X C-H	Define expansion for mode abbrev before point.
C-X -	Define expansion for global abbrev before point.
C-M-Space	Expand abbrev without inserting anything.
M-'	Mark a prefix to be glued to an abbrev following.
C-X U	Unexpand the last abbrev, or undo a C-X U.

M-X List Word Abbrevs<cr>
Shows definitions of all abbrevs.

M-X Edit Word Abbrevs<cr>
Lets you edit the definition list directly.

M-X Read Word Abbrev File♦<filename><cr>
Defines word abbrevs from a definition file.

M-X Write Word Abbrev File♦<filename><cr>
Makes a definition file from current abbrev definitions.

Readable Word Abbrev Files
Option variable to control abbrev file format.

This section describes the most common use of Word Abbrev mode. If you don't read any more than this, you can still use Word Abbrev mode quite effectively.

25.1.1. Adding Word Abbrevs

C-X C-A (^R Add Mode Word Abbrev) defines a mode abbrev for the word before point (this does not include any punctuation between that word and point, though). It prints the word before point in the echo area and ask you for that word's abbreviation. Type the abbrev (in the echo area, editing a little with Rubout and C-U) followed by a Return. The abbrev must be a "word": it must contain only letters and digits. If you'd rather define a global abbrev, use C-X + (^R Add Global Word Abbrev), which works similarly.

You can redefine an abbrev by C-X C-A or C-X +, but it tells you the old expansion and ask you to confirm the redefinition. Type Y or N.

To define an abbrev for more than one word of text, give C-X C-A or C-X + a numeric argument: an argument greater than 0 means the expansion is that many words before point; an argument of 0 means to use

the region (between point and mark). (By using the region specification you can make an abbrev for any text, not just a sequence of words.) The message in the echo area provides you with confirmation of just what the expansion will be; e.g. you might see:

```
Text Abbrev for "this is the expansion":
```

Sometimes you may think you already had an abbrev for some text, use it, and see that it didn't expand. In this case, the C-X C-H (^R Inverse Add Mode Word Abbrev) or C-X - (^R Inverse Add Global Word Abbrev) commands are helpful: they ask you to type in an *expansion* rather than an abbrev. In addition to defining the abbrev, they also expand it. If you give them a numeric argument, n, they use the nth word before point as the abbrev.

You can kill abbrevs (cause them to no longer expand) by giving a negative numeric argument to C-X C-A or C-X +. For instance, to kill the global abbrev "foo" type C-U - C-X + foo<cr>.

25.1.2. Controlling Abbrev Expansion

When an abbrev expands, the capitalization of the expansion is determined by the capitalization of the abbrev: If the abbrev is all lowercase, the expansion is as defined. If the abbrev's 1st letter is uppercase, the expansion's 1st letter is too. If the abbrev is all uppercase, there are two possibilities: if the expansion is a single word, it is all-uppercased; otherwise, each of its words has its first letter uppercased (e.g. for use in a title).

Abbrevs normally expand when you type some punctuation character; the abbrev expands and the punctuation character is inserted. There are other ways of expanding abbrevs: C-M-Space (^R Abbrev Expand Only) causes the abbrev just before point to be expanded without inserting any other character. M-' (^R Word Abbrev Prefix Mark) allows you to "glue" an abbrev onto any prefix: suppose you have the abbrev "committee" for "committee", and wish to insert "intercommittee "; type "inter", M-' (you will now see "inter-"), and then "committee "; "inter-comm " becomes "intercommittee ".

25.1.3. Unexpanding Abbrevs

C-X U (^R Unexpand Last Word) "unexpands" the last abbrev's expansion, replacing the last expansion with the abbrev that caused it. If any auto-filling was done because of the expansion (you had Auto Fill mode on), that too is undone. If you type another C-X U, the first one is "undone" and the abbrev is expanded again. (Sometimes you may find that C-X U unexpands an abbrev later than the one you're looking at. In this case, do another C-X U and go back and manually correct the earlier expansion. Only the last expansion can be undone.)

If you know beforehand that a word will expand, and want to prevent it, you can simply "quote" the punctuation character with C-Q. E.g. typing "comm", a C-Q, and then "." gives "comm." without expanding.

25.1.4. Listing Abbrevs

M-X List Word Abbrevs lists all currently defined abbrevs. An abbrev "foo" that expands to "this is an abbrev" in Text mode and has been expanded 3 (the "usage count") times, is listed as:

```
foo:     (Text)  3       "this is an abbrev"
```

An abbrev "gfoo" which expands to "this is a global abbrev" in all modes, expanded 11 times, is listed as:

```
gfoo:            11      "this is a global abbrev"
```

Note that any use of the double-quote character (") inside an expansion is doubled, to distinguish the use of " from the "s that surround the whole expansion. Thus if the global abbrev "helpk" expands to "the "Help" character", it is listed as:

```
helpk:           3       "the ""Help"" character"
```

25.1.5. Editing the Definition List

M-X Edit Word Abbrevs places you in a recursive editing level, editing the current word abbrev definition list. The abbrevs appear in the same format used by List Word Abbrevs. When you exit (via C-M-C), the current word abbrevs will be redefined from the edited definition list: any abbrevs that have been deleted from the list are killed, new ones added to the list are defined, and old ones changed are modified. In effect, after exiting the Edit Word Abbrev editing level, all previously-defined word abbrevs are killed, and the edited list is used to define new abbrevs. Typing C-] (Abort Recursive Edit) aborts Edit Word Abbrevs, without killing or redefining any abbrevs.

25.1.6. Saving Abbrev Definitions

M-X Write Word Abbrev File↵<filename><cr> writes an "abbrev definition file" which contains the definitions of all the abbrevs in your EMACS now. M-X Read Word Abbrev File↵<filename><cr> reads in such a file and defines the abbrevs. (Other abbrevs already defined are not affected unless the file redefines them.) If you don't supply a filename, the last one you used in either of these is used again, originally defaulting to WORDAB DEFNS. With these two commands, you can save the abbrevs you defined in one EMACS and restore them in another EMACS another day.

The format of the definition file is designed for fast loading, not ease of human readability. (But if you have to, you can figure it out enough to read or even edit it.) If you want M-X Write Word Abbrev File to write a human-readable version instead, set the option Readable Word Abbrev Files to 1. (M-X Read Word Abbrev File will be able to read this, but not as fast.)

If you have an init file, you might want to put TECO code like the following into it, in order to turn on Word Abbrev mode and have your abbrev definition file automatically read when EMACS starts up:

```
1m(m.mWord Abbrev Mode♦)
m(m.mRead Word Abbrev File♦)WORDAB DEFNS♦
```

25.2. Advanced Usage

The use of Word Abbrev mode as discussed in the previous section suffices for most users. However, some users who use Word Abbrev mode a lot or have highly tailored environments may desire more flexibility, or need more power to handle extreme situations, than the basic commands provide.

25.2.1. Alternatives and Customizations

M-X Make Word Abbrev♦<abbrev>♦<expansion>♦<mode><cr>

M-X Kill All Word Abbrevs<cr>

M-X Make These Characters Expand♦<characters><cr>

^R Kill Mode Word Abbrev

^R Kill Global Word Abbrev

Only Global Abbrevs
: Option variable if you only use globals.

Additional Abbrev Expanders
: Variable for adding a few more expanders.

WORDAB Ins Chars
: Variable for replacing entire set of expanders.

The basic commands for defining a new mode abbrev, C-X C-A (^R Add Mode Word Abbrev) and C-X C-H (^R Inverse Add Mode Word Abbrev), work only in the current mode. A more general command is M-X Make Word Abbrev which takes three string arguments: the first is the abbrev, the second is the expansion, and the third is the mode (e.g. "Text"). This command can also define global abbrevs, by

providing "*" as the mode name.

M-X Kill All Word Abbrevs<cr> is a very quick way of killing every abbrev currently defined. After this command, no abbrev will expand. (A slower but more careful way is with M-X Edit Word Abbrevs.)

The functions ^R Kill Mode Word Abbrev and ^R Kill Global Word Abbrev exist, but are not attached to any commands by default. If you want to avoid specifying negative arguments to C-X C-A (^R Add Mode Word Abbrev) and C-X + (^R Add Global Word Abbrev), you should attach these functions to characters. You could use the function M-X Set Key (See section 5.2 [Set Key], page 21.) to do this or have an init or EVARS file set them (See section 22.7 [Init], page 120.).

If you prefer to use only global abbrevs then you should set the option variable Only Global Abbrevs to 1. You can do this after or before turning on Word Abbrev mode; it makes no difference. This causes the global abbrev definers which would otherwise be on C-X + (^R Add Global Word Abbrev) and C-X - (^R Inverse Add Global Word Abbrev) to be on the easier to type characters C-X C-A and C-X C-H. In addition, the checking done whenever you type an expander character (e.g. the punctuation characters) is about three times faster (for the no-expansion case, which is what happens most of the time).

Normally, the following characters cause expansion (followed by whatever they would normally do were Word Abbrev mode off: e.g. insert themselves): !~@#;$%^&*-_=+[]()\|:'"{},<.>`/? and Space, Return, and Tab. You can, however, specify additional characters to cause expansion (digits, for instance, or greek letters on keyboards with Top-keys). M-X Make These Characters Expand♦<characters><cr> adds the characters in the string argument to the list of expanders. Alternatively, easier for init and EVARS files, you can set the variable Additional Abbrev Expanders to contain the string of characters. If you wish to completely replace the set of characters that cause expansion, set the variable WORDAB Ins Chars in your init file. See section 22.7 [init], page 120, for details on setting variables in init and EVARS files.

25.2.2. Manipulating Definition Lists

One reason some users have for manipulating the definition lists is to provide more structure to the definition environment than just the mode-global structure provided normally, e.g. to group together in a file those abbrevs pertaining to one topic.

M-X Insert Word Abbrevs<cr> inserts into the buffer a list of the current word abbrev definitions, in the format that M-X List Word Abbrevs uses.

M-X Define Word Abbrevs<cr> defines a set of word abbrevs from a definition list in the buffer. There should be nothing else besides the definition list in the buffer; or, if there is, you must *narrow* the buffer to just the definition list. See section 17 [Narrowing], page 77.

25.2.3. Dealing with Many Abbrevs

Some users rapidly build up a very large number of abbrevs. This causes a couple of problems that have to be dealt with in ways not discussed in the basic usage section: First, defining all those abbrevs when EMACS starts up can become too slow; this problem is discussed in the next section. Second, the commands that deal with the definition lists become unwieldy. The current release of the Word Abbrev mode package only has one command directly dealing with large definition lists:

M-X Word Abbrev Apropos♦<string><cr> shows you the definitions of just the abbrev definitions containing <string> (in the abbrev, in the mode, or in the expansion). The argument is actually a TECO search string (See section 19.3 [TECO search strings], page 85.). If you want to see the abbrevs which contain either <string1> *or* <string2>, separate the strings with a Control-O; e.g. to see abbrev definitions containing either "defn" or "wab", do M-X Word Abbrev Apropos♦defn↑Owab<cr>.

M-X Word Abbrev Apropos only works "approximately" though: it may miss some definitions (those defined by Edit Word Abbrevs, Define Word Abbrevs, or human-readable definition files), and it may also find more than you want in some rare cases. These problems are all related to efficiency questions.

25.2.4. Dumped EMACS Environments

M-X Write Word Abbrev File♦<filename><cr>
Writes a file of all abbrev definitions, before dumping.

M-X Read Word Abbrev File♦<filename><cr>
Reads file of abbrev definitions at init-time.

M-X Write Incremental Word Abbrev File♦<filename><cr>
Writes a file of changed abbrev definitions, since dumping.

M-X Read Incremental Word Abbrev File♦<filename><cr>
Reads file of changed abbrev definitions at startup-time.

Some users with highly customized EMACS environments (their init files take a long time to run) "dump out" their environments, in effect creating another EMACS-like program (the "dump") which starts up much faster. (For instance, 1.7 cpu seconds instead of 70.5 cpu seconds. See the file INFO;CONV >, for more details about dumping environments.) Since the dumped environment contains word abbrev definitions, a dumped environment with hundreds of abbrevs can start just as quickly as if it had none. (But reading all these abbrevs with M-X Read Word Abbrev File in the init file originally took a long time.) Thus for these users, it is important that at dump-startup time, only to read in those abbrevs which were changed or defined *since* the environment was dumped out. A file which contains only these new abbrev's definitions is called an

incremental word abbrev file. (It also can specify that certain abbrevs are to be killed if they were defined when the environment was dumped out, but subsequently killed.)

The startup for the dump should use the Read Incremental Word Abbrev File function instead of Read Word Abbrev File. It takes the filename as a string argument, which defaults to INCABS >. The command M-X Write Incremental Word Abbrev File♦<filename><cr> writes such a file, writing out those abbrevs more recent than the dump (including ones read by Read Incremental Word Abbrev File and ones defined in the current editing session). You might want to call this command after defining some abbrevs, before exiting EMACS. (Plans are afoot to have this be done automatically as an option.)

When you want to dump out a new EMACS, first create a new, complete word abbrev definition file using M-X Write Word Abbrev File. This file now has *all* abbrevs in it, and you can thus delete any incremental definition files you have. Then start up the new EMACS from scratch, using the init file, and dump it. (The init file in general should call Read Word Abbrev File and then *also* call Read Incremental Word Abbrev File, just in case there are both kinds of files around. The startup calls only Read Incremental Word Abbrev File.) Note that to handle the possibility of the incremental file not existing, the TECO code in the startup subroutine can look like (though soon this won't be necessary):

```
e?<filename>♦"e
    m(m.mRead Incremental Word Abbrev File♦)<filename>♦'
```

25.3. Teco Details for Extension Writers

This section documents some details that users programming extensions may need to know, in order to interact properly with Word Abbrev mode operation or to build upon it.

There are two hooks that you can provide: The variable WORDAB Setup Hook, if non-0, is executed when the WORDAB library is loaded and sets itself up. (M-X Word Abbrev Mode<cr> in the default EMACS environment auto-loads the WORDAB library.)

The variable Word Abbrev Hook, if non-0, is executed when Word Abbrev mode is turned on or off. It is passed a numeric argument which is positive if Word Abbrev mode is being turned on, and 0 or negative if it is being turned off. Also, if this hook exists, Word Abbrev mode does not redefine any characters; it assumes the hook will do that. Thus C-X C-A, C-X C-H, C-X +, C-X -, M-', C-M-Space, and C-X U will not be bound unless the hook binds them.

The abbrev definers, e.g. C-X C-A (^R Add Mode Word Abbrev), check to see if the volatile TECO mark, fs^RMark♦, is set; if it is, then the region between point and fs^RMark♦ is used as the expansion. The intention is to provide a mechanism for simple but safe expansion marking.

Finally, the general way that Word Abbrev mode works is this: at certain times, when characters are likely to have been reconnected, a Word Abbrev mode subroutine looks at each of the expander characters to see if they are running an expander or have been rebound. If they don't have expanders, they are reset to an expander function (which first checks for expansion and then calls the "old" function, what the character was bound to before). The problem is that it is not really possible to efficiently catch all the times that characters of interest are rebound. So, as a good guess, Word Abbrev mode looks at these characters when the & Set Mode Line function is called. This happens when major or minor modes change, when buffer switching happens, and when Set Key is used. These are the standard times that connections are changed. However, the extension writer must be careful about rebinding expander characters. If an extension might do this, it should do IfsMode Change♦ to cause expansions to be redefined.

26. The PICTURE Subsystem, an Editor for Text Pictures

If you want to create a picture made out of text characters (for example, a picture of the division of a register into fields, as a comment in a program), the PICTURE package can make it easier.

Do M-X Load Lib♦PICTURE<cr>, and then M-X Edit Picture is available. Do M-X Edit Picture with point and mark surrounding the picture to be edited. Edit Picture enters a recursive editing level (which you exit with C-M-C, as usual) in which certain commands are redefined to make picture editing more convenient.

While you are inside Edit Picture, all the lines of the picture are padded out to the margin with spaces. This makes two-dimensional motion very convenient; C-B and C-F move horizontally, and C-N and C-P move vertically without the inaccuracy of a ragged right margin. When you exit from Edit Picture, spaces at the ends of lines are removed. Nothing stops you from moving outside the bounds of the picture, but if you make any changes there slightly random things may happen.

Edit Picture makes alteration of the picture convenient by redefining the way printing characters and Rubout work. Printing characters are defined to replace (overwrite) rather than inserting themselves. Rubout is defined to undo a printing character: it replaces the previous character with a space, and moves back to it.

Return is defined to move to the beginning of the next line. This makes it usable for moving to the next apparently blank (but actually filled with nothing but spaces) line, just as you use Return normally with lines that are really empty. C-O creates new blank lines after point, but they are created full of spaces.

Tab is redefined to indent (by moving over spaces, not inserting them) to under the first non-space on the previous line. Linefeed is as usual equivalent to Return followed by Tab.

Four movement-control commands exist to aid in drawing vertical or horizontal lines: If you give the command M-X Up Picture Movement, each character you type thereafter will cause the cursor to move up instead of to the right. Thus if you want to draw a line of dashes up to some point, you can give the command Up Picture Movement, type enough dashes to make the line, and then give the command Right Picture Movement to put things back to normal. Similarly, there are functions to cause downward and leftward movement: Down Picture Movement and Left Picture Movement. These commands remain in effect only until you exit the Edit Picture function. (One final note: you can use these cursor movement commands outside of Edit Picture too, even when not in Overwrite mode. You have to be somewhat careful though.)

Possible future extensions include alteration of the kill and un-kill commands to replace instead of deleting

and inserting, and to handle rectangles if two corners are specified using point and the mark.

27. Sorting Functions

The SORT library contains functions called Sort Lines, Sort Paragraphs and Sort Pages, to sort the region alphabetically line by line, paragraph by paragraph or page by page. For example, Sort Lines rearranges the lines in the region so that they are in alphabetical order.

Paragraphs are defined in the same way as for the paragraph-motion functions (See section 11.2 [Paragraphs], page 44.) and pages are defined as for the page motion commands (See section 18 [Paging], page 79.). All of these functions can be undone by the Undo command (See section 24.3 [Undo], page 132.). They take no arguments.

Appendix I
Particular Types of Terminals

I.1. Ideal Keyboards

An ideal EMACS keyboard can be recognized because it has a Control key and a Meta key on each side, with another key labelled Top above them.

On an ideal keyboard, to type any character in the 9-bit character set, hold down Control or Meta as appropriate while typing the key for the rest of the character. To type C-M-K, type K while holding down Control and Meta.

You will notice that there is a key labeled "Escape" and a key labeled "Alt". The Altmode character is the one labeled "Alt". "Escape" has other functions entirely; it is handled by ITS and has nothing to do with EMACS. While we are talking about keys handled by ITS, on Meta keyboards the way to interrupt a program is CALL, rather than Control-Z, and entering communicate mode uses the BACK-NEXT key rather than Control-_. CALL *echoes* as ↑Z, but if you type C Z it is just an ordinary character which happens to be an EMACS command to return to the superior. Similarly, BACK-NEXT echoes as ↑_ but if you type ↑_ it is just an EMACS command which happens not to be defined.

The key labeled "Top" is an extra shift key. It is used to produce the peculiar "SAIL" graphics characters which appear on the same keys as the letters. The "Shift" key gets you upper-case letters, but "Top" gets you the SAIL characters. As EMACS commands, these characters are normally self-inserting, like all printing characters. But once inserted, SAIL characters are really the same as ASCII control characters, and since characters in files are just 7 bits there is no way to tell them apart. EMACS can display them either as ASCII control characters, using an uparrow or caret to indicate them, or it can display them as SAIL characters, whichever you like. The character Control-Alpha (Alpha is a SAIL character) toggles the choice. You can only type this command on a terminal with a Top key, but since only such terminals can display SAIL characters anyway this is no loss.

One other thing you can do with the Top key is type the Help character, which is Top-H on these keyboards. BACK-NEXT H also works, though.

For inserting an Altmode, on an ideal keyboard you can type C-M-Altmode. C-Altmode is a synonym for C-M-C (^R Exit).

The "bit prefix" characters that you must use on other terminals are also available on terminals with Meta keys, in case you find them more convenient or get into habits on those other terminals.

To type numeric arguments on these keyboards, type the digits or minus sign while holding down either Control or Meta.

I.2. Keyboards with an "Edit" key

Keyboards with Edit keys probably belong to Datamedia or Teleray terminals. The Edit and Control keys are a pair of shift keys. Use the Control key to type Control characters and the Edit key to type Meta characters. Thus, the 9-bit EMACS character C-M-Q is typed by striking the "Q" key while holding down "Edit" and "Control".

While the Edit key is a true independent bit which can be combined with anything else you can type, the Control key really means "ASCII control". Thus, the only Control characters you can type are those which exist in ASCII. This includes C-A through C-Y, C-[, C-@, C-\, and C-^. C-Z and C-_ can be typed on the terminal but they are intercepted by the operating system and therefore unavailable as EMACS commands. C-[is not available because its spot in ASCII is pre-empted by Altmode. The corresponding Meta commands are also hard to type. If you can't type C-; directly, then you also can't type C-M-; directly.

Though you can't type C-; directly, you can use the bit prefix character C-^ and type C-^ ;. Similarly, while you can't type C-M-;, you can use the Control-Meta prefix C-C and type C-C ;. Because C-^ is itself awkward, we have designed the EMACS command set so that the hard-to-type Control (non-Meta) characters are rarely needed.

In order to type the Help character you must actually type two characters, C-_ and H. C-_ is an escape character for ITS itself, and C-_ followed by H causes ITS to give the Help character as input to EMACS.

To type numeric arguments, it is best to type the digits or minus sign while holding down the Edit key.

I.3. ASCII Keyboards

An ASCII keyboard allows you to type in one keystroke only the command characters with equivalents in ASCII. No Meta characters are possible, and not all Control characters are possible either. The Control characters which you can type directly are C-A through C-Y, C-[, C-@, C-\, and C-^. C-Z and C-_ can be typed on the terminal but they are intercepted by the operating system and therefore unavailable as EMACS commands. C-[is not available because its spot in ASCII is pre-empted by Altmode.

Those characters which you can't type directly can be typed as two character sequences using the bit prefix characters Altmode, C-C and C-^. Altmode turns on the Meta bit of the character that follows it. Thus, M-A can be typed as Altmode A, and C-M-A as Altmode C-A. Altmode can be used to get almost all of the

characters that can't be typed directly. C-C can be used to type any Control-Meta character, including a few that Altmode can't be used for because the corresponding non-Meta character isn't on the keyboard. Thus, while you can't type C-M-; as Altmode Control-:, since there is no Control-; in ASCII, you can type C-M-; as C-C ;. The Control (non-Meta) characters which can't be typed directly require the use of C-^, as in C-^ < to get the effect of C-<. Because C-^ by itself is hard to type, the EMACS command set is arranged so that most of these non-ASCII Control characters are not very important. Usually they have synonyms which are easier to type. In fact, in this manual only the easier-to-type forms are usually mentioned.

In order to type the Help character you must actually type two characters, C-_ and H. C-_ is an escape character for ITS itself, and C-_ followed by H causes ITS to give the Help character as input to EMACS.

On ASCII keyboards, you can type a numeric argument by typing an Altmode followed by the minus sign and/or digits. Then comes the command for which the argument is intended. For example, type Altmode 5 C-N to move down five lines. If the command is a Meta command, it must have an Altmode of its own, as in Altmode 5 Altmode F to move forward five words.

Note to customizers: this effect requires redefining the Meta-digit commands, since the Altmode and the first digit amount to a Meta-digit character. The new definition is ^R Autoarg, and the redefinition is done by the default init file.

If you use numeric arguments very often, and even typing the Altmode is painful, you might enjoy using Autoarg mode, in which you can specify a numeric argument by just typing the digits. See section 4 [Arguments], page 17, for details.

I.4. Upper-case-only Terminals

On terminals lacking the ability to display or enter lower case characters, a special input and output case-flagging convention has been defined for editing files which contain lower case characters.

The customary escape convention is that a slash prefixes any upper case letter; all unprefixed letters are lower case (see below for the "lower case punctuation characters"). This convention is chosen because lower case is usually more frequent in files containing any lower case at all. Upper case letters are displayed with a slash ("/") in front. Typing a slash followed by a letter is a good way to insert an upper case letter. Typing a letter without a slash inserts a lower case letter. For the most part, the buffer will appear as if the slashes had simply been inserted (type /A and it inserts an upper case A, which displays as /A), but cursor-motion commands will reveal that the slash and the A are really just one character. Another way to insert an upper-case letter is to quote it with C-Q.

Note that this escape convention applies only to display of the buffer and insertion in the buffer. It does

not apply to arguments of commands (it is hardly ever useful for them, since case is ignored in command names and most commands' arguments). Case conversion is performed when you type commands into the minibuffer, but not when the commands are actually executed.

The ASCII character set includes several punctuation characters whose codes fall in the lower case range and which cannot be typed or displayed on terminals that cannot handle lower case letters. These are the curly braces ("{" and "}"), the vertical bar ("|"), the tilde ("~"), and the accent grave. Their upper case equivalents are, respectively, the square brackets ("[" and "]"), the backslash ("\"), the caret ("^"), and the atsign ("@"). For these punctuation characters, EMACS uses the opposite convention of that used for letters: the ordinary, upper case punctuations display as and are entered as themselves, while the lower case forms are prefixed by slashes. This is because the "lower case" punctuations are much less frequently used. So, to insert an accent grave, type "/@".

When the slash escape convention is in effect, a slash is displayed and entered as two slashes.

This slash-escape convention is not normally in effect. To turn it on, the TECO command "-1$" (minus one dollar sign, not Altmode!) must be executed. The easiest way to do this is to use the minibuffer: Altmode Altmode -1$ Altmode Altmode. To turn off the escape convention (for editing a file of all upper case), the command is 0$ (zero dollar sign), or Altmode Altmode 0$ Altmode Altmode. If you use such a bad terminal frequently, you can define yourself an EMACS extension, a command to turn slash-escape on and off.

The lower case editing feature is actually more flexible than described here. Refer to the TECO commands F$ and FS CASE♦, using M-X TECDOC, for full details.

1.5. The SLOWLY Package for Slow Terminals

The SLOWLY library is intended as an aid for people using display terminals at slow speeds. It provides means of limiting redisplay to smaller parts of the screen, and for turning off redisplay for a time while you edit.

To use SLOWLY, do M-X Load Library♦SLOWLY<cr> and if your terminal is a display operating at 1200 baud or less (or if its speed is unknown) SLOWLY will set up the commands described here.

Comments, bugs, and suggestions to RWK@MIT-MC

1.5.1. Brief Description

SLOWLY provides an alternate version of the incremental searching comands on C-S and C-R, ^R Edit Quietly on C-X Q, a way to shrink the screen at either the top or the bottom on M-O, and more flexibility in where minibuffers get displayed. If SLOWLY is loaded, it redefines these commands only if the terminal speed is 1200 baud or less.

1.5.2. SLOWLY Commands

The commands provided are:

M-O (^R Set Screen Size)

This function reduces the amount of the screen used for displaying your text, down to a few lines at the top or the bottom. If called without an argument, it will use the same size as last time (or 3 if it hasn't been called before). If given a positive argument, that is taken to be the number of lines to use at the top of the screen. If given a negative argument, it is taken to be the number of lines at the bottom of the screen. If given an argument of 0, it returns to the use of the entire screen. The section of the screen that is in use is (defaultly) delimited by a line of 6 dashes. This command sets the variable Short Display Size.

C-S (^R Slow Display I-Search)

This function is just like the usual incremental search, except if the search would run off the screen and cause a redisplay, it narrows the screen to use only a few lines at the top or bottom of the screen to do the redisplay in. When the search is exited, use of the full screen resumes. The size of the window used for the search is the value of the variable Slow Search Lines. If it is positive, it is number of lines at top of screen; if negative, it is the number of lines at bottom of screen. The default is 1. The variable Slow Search Separator contains the string used to show the end of the search window. By default it is six dashes. See section 10 [Search], page 41.

C-R (^R Slow Reverse Display I-Search)

This searches in backwards in the style of ^R Slow Display I-Search.

C-X Q (^R Edit Quietly)

This function enters a recursive editing level with redisplay inhibited. This means that your commands are carried out but the screen does not change. C-L with no argument redisplays. So you can update the screen when you want to. Two C-L's in a row clear the screen and redisplay. C-L with an argument repositions the window, as usual (See section 15 [C-L], page 71.). To exit and resume continuous redisplay, use C-M-C.

1.5.3. Minibuffers

SLOWLY provides control over how minibuffers display on your screen. The variable Minibuffer Size specifies how many lines it takes up. If this is made negative, the minibuffer will appear at the bottom of the

screen instead of the top. Thus one mode of operation which some people like is to use ^R Set Screen Size to set up to not use the bottom 3 lines of the screen, and set Minibuffer Size to -3. This will permanently reserve 3 lines at the bottom of the screen for the minibuffer. See section 23 [Minibuffer], page 127.

The variable Minibuffer Separator holds the string used to separate the minibuffer area from the rest of the screen. By default, this is six dashes.

SLOWLY installs its minibuffer by defining the variable MM & Minibuffer.

1.5.4. SLOWLY Options

The simplest way to run SLOWLY is to simply load it, and use the default key assignments, etc. Here is what SLOWLY sets up when simply loaded normally, *provided* your terminal is no faster than 1200 baud.

If you want SLOWLY not set up these things unless your terminal is running at 300 baud or slower (ugh!), set the variable SLOWLY Maximum Speed to the highest speed at which SLOWLY is desired. Put the following in your EMACS init file:

```
300 M.VSLOWLY Maximum Speed♦
```

If you don't like the command assignments set up by SLOWLY, you can override them by defining the variable SLOWLY Setup Hook before loading SLOWLY. The value should be TECO commands to define the command assignments you wish.

SLOWLY normally uses lines of six dashes to separate areas of the screen. You can tell it to use something else instead. Minibuffers use the value of Minibuffer Separator, searches use the value of Slow Search Separator. If one of these is unspecified (the variable does not exist), the value of Default Separator is used. The separator for small screen mode is always the value of Default Separator. If the value specified is the null string, a blank line is used. If the value specified is zero, nothing (not even a blank line) is used. This is useful for searches, since you aren't going to be doing any editing in the search window.

Even though SLOWLY does not redefine the commands on a fast terminal, you might wish to load it only on slow terminals to save address space the rest of the time. This can be done in an init file with

```
fsospeed♦-1200:"g m(m.mLoad Library♦)SLOWLY♦'
```

Appendix II
Use of EMACS from Printing Terminals

While EMACS was designed to be used from a display terminal, you can use it effectively from a printing terminal. You cannot, however, learn EMACS using one.

All EMACS commands have the same editing effect from a printing terminal as they do from a display. All that is different is how they try to show what they have done. EMACS attempts to make the same commands that you would use on a display terminal act like an interactive line-editor. It does not do as good a job as editors designed originally for that purpose, but it succeeds well enough to keep you informed of what your commands are accomplishing, provided you know what they are supposed to do and know how they would look on a display.

The usual buffer display convention for EMACS on a printing terminal is that the part of the current line before the cursor is printed out, with the cursor following (at the right position in the line). What follows the cursor on the line is not immediately visible, but normally you will have a printout of the original contents of the line a little ways back up the paper. For example, if the current line contains the word "FOOBAR", and the cursor is after the "FOO", just "FOO" would appear on the paper, with the cursor following it. Typing the C-F command to move over the "B" would cause "B" to be printed, so that you would now see "FOOB" with the cursor following it. All forward-motion commands that move reasonably short distances print out what they move over.

Backward motion is handled in a complicated way. As you move back, the terminal backspaces to the correct place. When you stop moving back and do something else, a linefeed is printed first thing so that the printing done to reflect subsequent commands does not overwrite the text you moved back over and become garbled by it. The Rubout command acts like backward motion, but also prints a slash over the character rubbed out. Other backwards deletion commands act like backward motion; they do not print slashes (it would be an improvement if they did).

One command is different on a printing terminal: C-L, which normally means "clear the screen and redisplay". With no argument, it retypes the entire current line. An argument tells it to retype the specified number of lines around the current line.

Unfortunately, EMACS cannot perfectly attain its goal of making the text printed on the current line reflect the current line in the buffer, and keeping the horizontal position of the cursor correct. One reason is that it is necessary for complicated commands to echo, but echoing them screws up the "display". The only solution is to type a C-L whenever you have trouble following things in your mind. The need to keep a

mental model of the text being edited is, of course, the fundamental defect of all printing terminal editors.

Note: it is possible to make a specific command print on a printing terminal in whatever way is desired, if that is worth while. For example, Linefeed knows explicitly how to display itself, since the general TECO redisplay mechanism isn't able to handle it. Suggestions for how individual commands can display themselves are welcome, as long as they are algorithmic rather than simply of the form "please do the right thing".

Glossary

Aborting — Aborting a recursive editing level (q.v.) means canceling the command which invoked the recursive editing. For example, if you abort editing a message to be sent, the message is not sent. Aborting is done with the command C-]. See section 24.1 [Aborting], page 129.

Altmode — Altmode is a character, labelled Escape on some keyboards. It is the bit prefix character (q.v.) used to enter Meta-characters when the keyboard does not have a Meta (q.v.) key. See section 2 [Characters], page 9. Also, it delimits string arguments to extended commands. See section 5 [Extended], page 19.

Balance Parentheses — EMACS can balance parentheses manually or automatically. You can ask to move from one parenthesis to the matching one. See section 20.6.1 [Lists], page 94. When you insert a close parenthesis, EMACS can show the matching open. See section 20.4 [Matching], page 90.

Bit Prefix Character — A bit prefix character is a command which combines with the next character typed to make one character. They are used for effectively typing commands which the keyboard being used is not able to send. For example, to use a Meta-character when there is no Meta key on the keyboard, the bit prefix character Altmode (q.v.) is needed. See section 2 [Bit Prefix], page 9.

Buffer — The buffer is the basic editing unit; one buffer corresponds to one piece of text being edited. You can have several buffers, but at any time you are editing only one, the "selected" buffer, though two can be visible when you are using two windows. See section 14 [Buffers], page 67.

C- — C is an abbreviation for Control, in the name of a character. See section 2 [C-], page 9.

C-M- — C-M- is an abbreviation for Control-Meta, in the name of a character. See section 2 [C-M-], page 9.

Command — A command is a character or sequence of characters which, when typed by the user, fully specifies one action to be performed by EMACS. For example, "X" and "Control-F" and "Meta-X Text Mode<cr>" are commands. See section 2 [Command], page 9. Sometimes the first character of a multi-character command is also considered a command: M-X Text Mode<cr> is a command (an extended command), and M-X is also a command (a command to read a function name and invoke the function). See section 5 [Extended], page 19.

Completion — Completion is what EMACS does when it automatically fills out the beginning of an extended command name into the full name, or as much of it as can be deduced for certain. Completion occurs when Altmode, Space or Return is typed. See section 5 [Completion], page 19.

Connected — A character command in EMACS works by calling a function which it is "connected" to. Customization often involves connecting a character to a different function. See "Dispatch table". See section 2 [Connected], page 9.

Continuation Line
: When a line of text is longer than the width of the screen, it is displayed on more than one line of screen. We say that the line is continued, and that all screen lines used but the first are called continuation lines. See section 1 [Continuation], page 5.

Control
: Control is the name of a bit which each command character does or does not contain. A character's name includes the word Control if the Control bit is part of that character. Ideally, this means that the character is typed using the Control key: Control-A is typed by typing "A" while holding down Control. On most keyboards the Control key works in only some cases; the rest of the time, a bit prefix character (q.v.) must be used. See section 2 [Control], page 9.

Control-Character
: A Control character is a character which includes the Control bit.

Control-X Command
: A Control-X command is a two-character command whose first character is the prefix character Control-X. See section 2 [Control-X Command], page 9.

<cr>
: <cr> stands for the carriage return character, in contexts where the word "Return" might be confusing. See section 2 [<cr>], page 9.

CRLF
: CRLF stands for the sequence of two characters, carriage return followed by linefeed, which is used to separate lines in files and text being edited in EMACS. See section 2 [CRLF], page 9.

Cursor
: The cursor is the object on the screen which indicates the position called "point" (q.v.) at which insertion and deletion takes place. The cursor is part of the terminal, and often blinks or underlines the character where it is located. See section 3 [Cursor], page 13.

Customization
: Customization is making minor changes in the way EMACS works. It is often done by setting variables (See section 22.3 [Variables], page 114.) or by reconnecting commands (See section 5.2 [Reconnect], page 21.).

DEFUN
: A DEFUN is a list at the top level of list structure in a Lisp program. It is so named because most such lists are calls to the Lisp function DEFUN. See section 20.6.2 [], page 95.

Delete
: This is the label used on some terminals for the Rubout character.

Deletion
: Deletion means erasing text without saving it. EMACS deletes text only when it is expected not to be worth saving (all whitespace, or only one character). The alternative is "killing" (q.v.). See section 9.1 [Deletion], page 35.

Dispatch Table
: The dispatch table is what records the connections (q.v.) from command characters to functions. Think of a telephone switchboard connecting incoming lines (commands) to telephones (functions). A standard EMACS has one set of connections; a customized EMACS may have different connections. See section 5.2 [Dispatch Table], page 21.

Echo Area
: The echo area is the bottom three lines of the screen, used for echoing the arguments to commands, for asking questions, and printing brief messages. See section 1 [Echo Area], page 5.

Echoing — Echoing is acknowledging the receipt of commands by displaying them (in the echo area). Most programs other than EMACS echo all their commands. EMACS never echoes single-character commands; longer commands echo only if you pause while typing them.

Escape — Escape is the label used on some terminals for the Altmode character.

Exiting — Exiting EMACS means returning to EMACS's superior, normally HACTRN. See section 6.3 [Exiting], page 26. Exiting a recursive editing level (q.v.) means allowing the command which invoked the recursive editing to complete normally. For example, if you are editing a message to be sent, and you exit, the message is sent.

Extended Command — An extended command is a command which consists of the character Meta-X followed by the command name (really, the name of a function (q.v.)). An extended command requires several characters of input, but its name is made up of English words, so it is easy to remember. See section 5 [Extended], page 19.

Extension — Extension means making changes to EMACS which go beyond the bounds of mere customization. If customization is moving the furniture around in a room, extension is building new furniture. See the file INFO;CONV >.

Filling — Filling text means moving text from line to line so that all the lines are approximately the same length. See section 11.4 [Filling], page 47.

Function — A function is a named subroutine of EMACS. When you type a command, EMACS executes a function which corresponds to the command, and the function does the work. Short commands are connected to functions through the dispatch table (q.v.). Extended commands contain the name of the function to be called; this allows you to call any function. See section 5 [Extended], page 19.

Grinding — Grinding means reformatting a program so that it is indented according to its structure. See section 20.7 [Grinding], page 96.

Help — You can type the Help character at any time to ask what options you have, or to ask what any command does. See section 7 [Help], page 29.

Home Directory — Your home directory is the one on which your mail and your init files are stored.

INFO — INFO is the subsystem for perusing tree-structured documentation files. The documentation in INFO includes a version of the EMACS manual.

ITS — ITS is the Incompatible Timesharing System written at the MIT Artificial Intelligence Lab. EMACS was first developed on this system. Just what it is incompatible with has changed from year to year.

Kill Ring — The kill ring is where killed text is saved. It holds the last nine or so blocks of killed text. It is called a ring because you can bring any of the saved blocks to the front by rotating the ring. See section 9.1 [Kill ring], page 35.

Killing — Killing means erasing text and saving it inside EMACS to be recovered later if desired. Most EMACS commands to erase text do killing, as opposed to deletion (q.v.). See section 9.1 [Killing], page 35.

List — A list is, approximately, a text string beginning with an open parenthesis and ending with the matching close parenthesis. See section 20.6.1 [Lists], page 94. Actually there are a few complications to the syntax, which is controlled by the syntax table (See section 22.4 [Syntax], page 115.).

M- — M- in the name of a character is an abbreviation for Meta.

M-X — M-X is the character which begins an extended command (q.v.). Extended commands have come to be known also as "M-X commands", and an individual extended command is often referred to as "M-X such-and such". See section 5 [M-X], page 19.

Major Mode — The major modes are a mutually exclusive set of options which configure EMACS for editing a certain sort of text. Ideally, each programming language has its own major mode. See section 20 [Major Mode], page 87.

Mark — The mark points, invisibly, to a position in the text. Many commands operate on the text between point and the mark (known as "the region", q.v.). See section 8 [Mark], page 31.

Meta — Meta refers to the Meta key. A character's name includes the word Meta if the Meta key must be held down in order to type the character. If there is no Meta key, then the Altmode character is used as a prefix instead. See section 2 [Meta], page 9.

Meta Character — A Meta character is one whose character code includes the Meta bit. These characters can be typed only by means of a Meta key or by means of the Metizer command (q.v.).

Metizer — The metizer is another term for the bit prefix character for the Meta bit; namely, Altmode (q.v.).

Minibuffer — The minibuffer is a facility for editing and then executing a TECO program. See section 23 [Minibuffer], page 127.

Minor mode — A minor mode is an optional feature of EMACS which can be switched on or off independently of all other features. Each minor mode is both the name of an option (q.v.) and the name of an extended command to set the option. See section 22.1 [Minor Mode], page 111.

MM-command — This is an obsolete synonym for "extended command".

Mode line — The mode line is a line just above the echo area (q.v.), used for status information. See section 1.1 [Mode Line], page 6.

Narrowing — Narrowing means limiting editing to only a part of the text in the buffer. Text outside that part is inaccessible to the user until the boundaries are widened again, but it is still there, and saving the file saves it all. See section 17 [Narrowing], page 77.

Numeric Argument — A numeric argument is a number specified before a command to change the effect of the command. Often the numeric argument serves as a repeat count. See section 4 [Numeric Argument], page 17.

Option — An option is a variable which exists to be set by the user to change the behavior of EMACS commands. This is an important method of customization. See section 22.3 [Options], page 114.

Parse — We say that EMACS parses words or expressions in the text being edited. Really, all it knows how to do is find the other end of a word or expression. See section 22.4 [Syntax], page 115.

Point — Point is the place in the buffer at which insertion and deletion occur. Point is considered to be between to characters, not at one character. The terminal's cursor (q.v.) indicates the location of point. See section 3 [Point], page 13.

Prefix Character — A prefix character is a command whose sole function is to introduce a set of multi-character commands. Control-X (q.v.) is a prefix character. The bit prefix characters (q.v.) are other examples.

Prompt — A prompt is text printed in the echo area to ask the user for input. Printing a prompt is called "prompting". EMACS can prompt when a command requires an argument, or when only part of a command has been typed. However, the prompt will not appear unless you pause in your typing. See section 5 [Prompt], page 19.

Q-Registers — Q-registers are internal TECO variables which can be used by EMACS or by the user to store text or numbers.

Quitting — Quitting means interrupting a command which is partially typed in or already executing. It is done with Control-G. See section 24.1 [Quitting], page 129.

Quoting — Quoting means depriving a character of its usual special significance. It is usually done with Control-Q. What constitutes special significance depends on the context and on convention. For example, an "ordinary" character as an EMACS command inserts itself; so you can insert any other character, such as Rubout, by quoting it as in Control-Q Rubout. Not all contexts allow quoting.

Recursive Editing Level — A recursive editing level is a state in which part of the execution of a command involves asking the user to edit some text. This text may or may not be the same as the text to which the command was applied. The mode line indicates recursive editing levels with square brackets ("[" and "]"). See section 6.2 [Recursive Editing Level], page 26.

Redisplay — Redisplay is the process of correcting the image on the screen to correspond to changes that have been made in the text being edited. See section 1 [Redisplay], page 5.

Region — The region is the text between point (q.v.) and the mark (q.v.). The terminal's cursor indicates the location of point, but the mark is invisible. Many commands operate on the text of the region. See section 8 [Region], page 31.

Return — Return is the carriage return character, used as input to EMACS. Return is used as a command in itself to insert a line separator. It also terminates arguments for most commands. See section 2 [Return], page 9.

Rubout — Rubout is a character, sometimes labelled "Delete". It is used as a command to delete one character of text. It also deletes one character when an EMACS command is reading an argument.

S expression — An s-expression is the basic syntactic unit of Lisp: either a list, or a symbol containing no parentheses (actually, there are a few exceptions to the rule, based on the syntax of Lisp). See section 20.6.1 [S-expressions], page 94.

Selecting — Selecting a buffer (q.v.) means making editing commands apply to that buffer as opposed to any other. At all times one buffer is selected and editing takes place in that buffer. See section 14 [Select], page 67.

Self-documentation — Self-documentation is the feature of EMACS which can tell you what any command does, or give you a list of all commands related to a topic you specify. You ask for self-documentation with the Help character. See section 7 [Self-documentation], page 29.

String Argument — A string argument is an argument which follows the command name in an extended command. In "M-X Apropos♦word<cr>", "Word" is a string argument to the Apropos command. See section 5 [String Arguments], page 19.

Subsystem — A subsystem of EMACS is an EMACS command which, itself, reads commands and displays the results. Examples are INFO, which is for perusing documentation; DIRED, which is for editing directories; RMAIL and BABYL, which are for reading and editing mail. The word "subsystem" implies that it offers many independent commands which can be used freely. If an EMACS function asks specific questions, we do not call it a subsystem.

Usually the subsystem continues in operation until a specific command to exit (usually "Q") is typed. The commands for a subsystem do not usually resemble ordinary EMACS commands, since editing text is not their purpose. The Help character should elicit the subsystem's documentation. See section 6.1 [Subsystems], page 25.

Syntax Table — The syntax table tells EMACS which characters are part of a word, which characters balance each other like parentheses, etc. See section 22.4 [Syntax], page 115.

Tailoring — This is a synonym for customization (q.v.).

TECO Search String — A TECO search string is a sort of pattern used by the TECO search command, and also by various EMACS commands which use the TECO search command. See section 19.3 [TECO search string], page 85.

Top Level — Top level is the normal state of EMACS, in which you are editing the text of the file you have visited. You are at top level whenever you are not in a recursive editing level or a subsystem (q.v.).

Twenex — Twenex is the operating system which DEC likes to call "TOPS-20". However, a person should not be forced to call a system "tops" unless he really thinks so. Come now, DEC, don't you think people will praise your products voluntarily? The name "Twenex" is also more appropriate because Twenex was developed from the Tenex system, and has no relationship to "TOPS-10". What's more, it's very euphonious.

Typeout — Typeout is a message printed by an EMACS command, which overwrites the area normally used for displaying the text being edited, but which does not become part of the text. Typeout is used for messages which might be too long to fit in the echo area (q.v.). See section 1 [Typeout], page 5.

Undo — Undo is a command which undoes the effect on the buffer of a previous command. Only some commands are undoable and only the most recent undoable command can be undone. See section 24.3 [Undo], page 132.

Un-killing — Un-killing means reinserting text previously killed. It can be used to undo a mistaken kill, or for copying or moving text. See section 9.2 [Un-killing], page 37.

User Name — Your user name is the name you use to log in. It identifies you as opposed to all the other users. It may be the same as your home directory's name.

Variable — A variable is a name with which EMACS associates a value, which can be a number or a string. See section 22.3 [Variables], page 114. Some variables ("options") are intended to be used or set by the user; others are for purely internal purposes.

Virtual Boundaries — The virtual boundaries delimit the accessible part of the buffer, when narrowing (q.v.) is in effect. See section 17 [Virtual Boundaries], page 77.

Visiting — Visiting a file means loading its contents into a buffer (q.v.) where they can be edited. See section 13.1 [Visiting], page 55.

Wall Chart — The wall chart is a very brief EMACS reference sheet giving one line of information about each short command. A copy of the wall chart appears in this manual Whitespace
Whitespace is any run of consecutive formatting characters (space, tab, carriage return, linefeed, and backspace).

Widening — Widening is the operation which undoes narrowing (q.v.). See section 17 [Widening], page 77.

Window — A window is a region of the screen in which text being edited is displayed. EMACS can support two windows. See section 16 [Windows], page 73. "The window" also means the position in the buffer which is at the top of the screen. See section 15 [The Window], page 71.

Working Directory — This is the directory which you have told the system you wish to operate on primarily at the moment. Often this will be the same as your home directory (q.v.). It is specified with the DDT command :CWD <directory>.

^R — The string "^R" is the beginning of many function names. See section 5.2 [^R], page 21.

^R mode — ^R mode is the real time editing mode of TECO. EMACS *always* operates in this mode.

Command Index

This index contains brief descriptions with cross references for all commands, grouped by topic. Within each topic, they are in alphabetical order. Our version of alphabetical order places non-control non-meta characters first, then control characters, then meta characters, then control-meta characters. Control-X comes last.

Prefix Characters

Altmode (^R Prefix Meta)
Altmode is a bit prefix character which turns on the Meta bit in the next character. Thus, Altmode F is equivalent to the single character Meta-F, which is useful if your keyboard has no Meta key. See section 2 [Altmode], page 9.

Control-^ (^R Prefix Control)
Control-^ is a bit prefix character which turns on the Control bit in the following character. Thus, Control-^ < is equivalent to the single character Control-<. See section 2 [Control-^], page 9.

Control-C (^R Prefix Control-Meta)
Control-C is a bit prefix character which turns on the Control bit and the Meta bit in the following character. Thus, Control-C ; is equivalent to the single character Control-Meta-; . See section 2 [Control-C], page 9.

Control-Q (^R Quoted Insert)
Control-Q inserts the following character. This is a way of inserting control characters. See section 3 [Control-Q], page 13.

Control-U (^R Universal Argument)
Control-U is a prefix for numeric arguments which works the same on all terminals. See section 4 [Control-U], page 17.

Control-X
Control-X is a prefix character which begins a two-character command. Each combination of Control-X and another character is a "Control-X command". Individual Control-X commands appear in this index according to their uses.

Meta-X (^R Extended Command)
Meta-X is a prefix character which introduces an extended command name. See section 5 [Meta-X], page 19.

Control-Meta-X (^R Instant Extended Command)
Control-Meta-X is another way of invoking an extended command. Instead of putting the arguments in the same line as the command name, the command reads the arguments itself. See section 5 [Control-Meta-X], page 19.

Control-digits, Meta-digits, Control-Meta-digits

These all specify a numeric argument for the next command. See section 4 [Arguments], page 17.

Control-Minus, Meta-Minus, Control-Meta-Minus
: These all begin a negative numeric argument for the next command. See section 4 [Arguments], page 17.

Simple Cursor Motion

Control-A (^R Beginning of Line, built-in function)
: Control-A moves to the beginning of the line. See section 3 [Control-A], page 13.

Control-B (^R Backward Character, built-in function)
: Control-B moves backward one character. See section 3 [Control-B], page 13.

Control-E (^R End of Line, built-in function)
: Control-E moves to the end of the line. See section 3 [Control-E], page 13.

Control-F (^R Forward Character, built-in function)
: Control-F moves forward one character. See section 3 [Control-F], page 13.

Control-H (^R Backward Character, built-in function)
: Control-H moves backward one character. See section 3 [Control-H], page 13.

Control-N (^R Down Real Line)
: Control-N moves vertically straight down. See section 3 [Control-N], page 13.

Control-P (^R Up Real Line)
: Control-P moves vertically straight up. See section 3 [Control-P], page 13.

Control-R (^R Reverse Search)
: Control-R is like Control-S but searches backward. See section 10 [Control-R], page 41.

Control-S (^R Incremental Search)
: Control-S searches for a string, terminated by Altmode. It searches as you type. See section 10 [Control-S], page 41.

Meta-< (^R Goto Beginning)
: Meta-< moves to the beginning of the buffer. See section 3 [Meta-<], page 13.

Meta-> (^R Goto End)
: Meta-> moves to the end of the buffer. See section 3 [Meta->], page 13.

Control-X Control-N (^R Set Goal Column)
: Control-X Control-N sets a horizontal goal for the Control-N and Control-P commands. When there is a goal, those commands try to move to the goal column instead of straight up or down.

Lines

Return (^R CRLF)
: Return inserts a line separator, or advances onto a following blank line. See section 3 [Return], page 13.

Control-O (^R Open Line, built-in function)
: Control-O inserts a line separator, but point stays before it. See section 3 [Control-O], page 13.

Control-X Control-O (^R Delete Blank Lines)
: Control-X Control-O deletes all but one of the blank lines around point. If the current line is not blank, all blank lines following it are deleted. See section 3 [Control-X Control-O], page 13.

Control-X Control-T (^R Transpose Lines)
: Control-X Control-T transposes the contents of two lines. See section 12 [Control-X Control-T], page 53.

Killing and Un-killing

Rubout (^R Backward Delete Character, built-in function)
: Rubout deletes the previous character. See section 3 [Rubout], page 13.

Control-Rubout (^R Backward Delete Hacking Tabs, built-in function)
: Control-Rubout deletes the previous character, but converts a tab character into several spaces. See section 20.6 [Control-Rubout], page 93.

Control-D (^R Delete Character, built-in function)
: Control-D deletes the next character. See section 3 [Control-D], page 13.

Control-K (^R Kill Line)
: Control-K kills to the end of the line, or, at the end of a line, kills the line separator. See section 9.1 [Control-K], page 35.

Control-W (^R Kill Region)
: Control-W kills the region, the text betwen point and the mark. See section 9.1 [Control-W], page 35. See section 8 [Region], page 31.

Control-Y (^R Un-kill)
: Control-Y reinserts the last saved block of killed text. See section 9.2 [Control-Y], page 37.

Meta-W (^R Copy Region)
: Meta-W saves the region as if it were killed without removing it from the buffer. See section 9.2 [Meta-W], page 37.

Meta-Y (^R Un-kill Pop)
: Meta-Y rolls the kill ring to reinsert saved killed text older than the most recent kill. See section 9.2 [Meta-Y], page 37.

Control-Meta-W (^R Append Next Kill)
Control-Meta-W causes an immediately following kill command to append its text to the last saved block of killed text. See section 9.2 [Control-Meta-W], page 37.

Control-X T (^R Transpose Regions)
Control-X T transposes two arbitrary regions defined by point and the last three marks. See section 12 [Control-X T], page 53.

Scrolling and Display Control

Control-L (^R New Window)
Control-L clears the screen and centers point in it. With an argument, it can put point on a specific line of the screen. See section 15 [Control-L], page 71.

Control-V (^R Next Screen)
Control-V scrolls downward by a screenful or several lines. See section 15 [Control-V], page 71.

Meta-R (^R Move to Screen Edge)
Meta-R moves point to beginning of the text on a specified line of the screen. See section 15 [Meta-R], page 71.

Meta-V (^R Previous Screen)
Meta-V scrolls upward by a screenful or several lines. See section 15 [Meta-V], page 71.

Control-Meta-R (^R Reposition Window)
Control-Meta-R tries to center on the screen the function or paragraph you are looking at. See section 15 [Control-Meta-R], page 71.

Control-Meta-V (^R Scroll Other Window)
Control-Meta-V scrolls the other window up or down, when you are in two window mode. See section 16 [Control-Meta-V], page 73.

The Mark and the Region

Control-< (^R Mark Beginning)
Control-< sets the mark at the beginning of the buffer. See section 8 [Control-<], page 31.

Control-> (^R Mark End)
Control-> sets the mark at the end of the buffer. See section 8 [Control->], page 31.

Control-@ (^R Set/Pop Mark)
Control-@ sets the mark or moves to the location of the mark. See section 8 [Control-@], page 31.

Meta-@ (^R Mark Word)
Meta-@ puts the mark at the end of the next word. See section 11.1 [Meta-@], page 43.

Meta-H (^R Mark Paragraph)

Meta-H puts point at the beginning of the paragraph and the mark at the end. See section 11.2 [Meta-H], page 44.

Control-Meta-@ (^R Mark Sexp)
Control-Meta-@ puts the mark at the end of the next s-expression. See section 20.6.1 [Control-Meta-@], page 94.

Control-Meta-H (^R Mark DEFUN)
Control-Meta-H puts point at the beginning of the current DEFUN and the mark at the end. See section 20.6.2 [Control-Meta-H], page 95.

Control-X H (^R Mark Whole Buffer)
Control-X H puts point at the beginning of the buffer and the mark at the end. See section 8 [Control-X H], page 31.

Control-X Control-P (^R Mark Page)
Control-X Control-P puts point at the beginning of the current page and the mark at the end. See section 18 [Control-X Control-P], page 79.

Control-X Control-X (^R Exchange Point and Mark)
Control-X Control-X sets point where the mark was and the mark where point was. See section 8 [Control-X Control-X], page 31.

Whitespace and Indentation

Tab (^R Indent According to Mode)
Tab either adjusts the indentation of the current line or inserts some indentation, in a way that depends on the major mode. See section 11.3 [Indentating Text], page 46. See section 20.3 [Indentating Programs], page 89.

Linefeed (^R Indent New Line)
Linefeed is equivalent to Return followed by Tab. It moves to a new line and indents that line. If done in the middle of a line, it breaks the line and indents the new second line. See section 11.3 [Linefeed], page 46.

Meta-Tab (^R Tab to Tab Stop)
Meta-Tab indents to the next EMACS-defined tab stop. See section 11.3 [Meta-Tab], page 46.

Meta-M (^R Back to Indentation)
Meta-M positions the cursor on the current line after any indentation. See section 11.3 [Meta-M], page 46.

Meta-\ (^R Delete Horizontal Space)
Meta-\ deletes all spaces and tab characters around point. See section 11.3 [Meta-\], page 46.

Meta-^ (^R Delete Indentation)
Meta-^ joins two lines, replacing the indentation of the second line with zero or one space, according to the context. See section 11.3 [Meta-^], page 46.

Control-Meta-O (^R Split Line)
Control-Meta-O breaks a line, preserving the horizontal position of the second half by indenting it to its old starting position. See section 11.3 [Control-Meta-O], page 46.

Control-Meta-\ (^R Indent Region)
Control-Meta-\ indents each line in the region, either by applying Tab to each line, or by giving each the same specified amount of indentation. See section 11.3 [Control-Meta-\], page 46.

Control-X Tab (^R Indent Rigidly)
Control-X Tab shifts all the lines in the region right or left the same number of columns. See section 11.3 [Control-X Tab], page 46.

Words, Sentences and Paragraphs

Control-X Rubout (^R Backward Kill Sentence)
Control-X Rubout kills back to the beginning of the sentence. See section 11.2 [Control-X Rubout], page 44.

Meta-A (^R Backward Sentence)
Meta-A moves to the beginning of the sentence. See section 11.2 [Meta-A], page 44.

Meta-B (^R Backward Word)
Meta-B moves backward one word. See section 11.1 [Meta-B], page 43.

Meta-D (^R Kill Word)
Meta-D kills one word forward. See section 11.1 [Meta-D], page 43.

Meta-E (^R Forward Sentence)
Meta-E moves to the end of the sentence. See section 11.2 [Meta-E], page 44.

Meta-F (^R Forward Word)
Meta-F moves forward one word. See section 11.1 [Meta-F], page 43.

Meta-H (^R Mark Paragraph)
Meta-H puts point at the front of the current paragraph and the mark at the end. See section 11.2 [Meta-H], page 44.

Meta-K (^R Kill Sentence)
Meta-K kills to the end of the sentence. See section 11.2 [Meta-K], page 44.

Meta-T (^R Transpose Words)
Meta-T transposes two consecutive words. See section 11.1 [Meta-T], page 43.

Meta-[(^R Backward Paragraph)
Meta-[moves to the beginning of the paragraph. See section 11.2 [Meta-[], page 44.

Meta-] (^R Forward Paragraph)
Meta-] moves to the end of the paragraph. See section 11.2 [Meta-]], page 44.

Meta-Rubout (^R Backward Kill Word)

Meta-Rubout kills the previous word. See section 11.1 [Meta-Rubout], page 43.

Filling Text

Meta-G (^R Fill Region)
Meta-G fills the region, treating it (usually) as one paragraph. See section 11.4 [Meta-G], page 47.

Meta-Q (^R Fill Paragraph)
Meta-Q fills the current or next paragraph. See section 11.4 [Meta-Q], page 47.

Meta-S (^R Center Line)
Meta-S centers the current line. See section 11.4 [Meta-S], page 47.

Control-X : (^R Set Fill Prefix)
Control-X : specifies the fill prefix, which is used for filling indented text. See section 11.4 [Control-X Colon], page 47.

Control-X F (^R Set Fill Column)
Control-X F sets the variable Fill Column which controls the margin for filling and centering. See section 11.4 [Control-X F], page 47.

Exiting

Control-] (Abort Recursive Edit)
Control-] aborts a recursive editing level; that is to say, exits it without allowing the command which invoked it to finish. See section 24.1 [Control-]], page 129.

Control-Meta-C (^R Exit, built-in function)
Control-Meta-C exits from a recursive editing level and allows the command which invoked the recursive editing level to finish. At top level, it exits from EMACS to its superior job. See section 6.3 [Control-Meta-C], page 26.

Control-X Control-C (^R Return to Superior)
Control-X Control-C returns from EMACS to its superior job, even if EMACS is currently inside a recursive editing level. In that case, re-entering EMACS will find it still within the recursive editing level. See section 6.3 [Control-X Control-C], page 26.

Pages

Control-X L (^R Count Lines Page)
Control-X L prints the number of lines on the current page, and how many come before point and how many come after. See section 18 [Control-X L], page 79.

Control-X P (^R Set Bounds Page)
Control-X P narrows the virtual boundaries to the current page. See section 17 [Control-X P], page 77.

Control-X [(^R Previous Page)
: Control-X [moves backward to the previous page boundary. See section 18 [Control-X []. page 79.

Control-X] (^R Next Page)
: Control-X] moves forward to the next page boundary. See section 18 [Control-X]], page 79.

Control-X Control-P (^R Mark Page)
: Control-X Control-P puts point at the beginning and the mark at the end of the current page. See section 18 [Control-X Control-P], page 79.

Lisp

Meta-([^R Make ()]
: Meta-(places a pair of parentheses around the next several s-expressions. See section 20.6.1 [Meta-(], page 94.

Meta-) [^R Move Over)]
: Meta-) moves past the next close parenthesis and adjusts the indentation of the following line. See section 20.6.1 [Meta-)], page 94.

Control-Meta-Tab (^R Indent for Lisp)
: Control-Meta-Tab adjusts the indentation of the current line for proper Lisp style. See section 20.3 [Control-Meta-Tab], page 89.

Control-Meta-((^R Backward Up List)
: Control-Meta-(moves backward up one level of list structure. See section 20.6.1 [Control-Meta-(], page 94.

Control-Meta-) (^R Up List)
: Control-Meta-) moves forward up one level of list structure. See section 20.6.1 [Control-Meta-)], page 94.

Control-Meta-@ (^R Mark Sexp)
: Control-Meta-@ puts the mark at the end of the next s-expression. See section 8 [Control-Meta-@], page 31.

Control-Meta-A (^R Beginning of DEFUN)
: Control-Meta-A moves to the beginning of the current DEFUN. See section 20.6.2 [Control-Meta-A], page 95.

Control-Meta-B (^R Backward Sexp)
: Control-Meta-B moves backward over one s-expression. See section 20.6.1 [Control-Meta-B], page 94.

Control-Meta-D (^R Down List)
: Control-Meta-D moves forward and down a level in list structure. See section 20.6.1 [Control-Meta-D], page 94.

Control-Meta-E (^R End of DEFUN)

Control-Meta-E moves to the end of the current DEFUN. See section 20.6.2 [Control-Meta-E], page 95.

Control-Meta-F (^R Forward Sexp)
Control-Meta-F moves forward over one s-expression. See section 20.6.1 [Control-Meta-F], page 94.

Comtrol-Meta-G (^R Format Code)
Control-Meta-G grinds the s-expression after point. See section 20.7 [Control-Meta-G], page 96.

Control-Meta-H (^R Mark DEFUN)
Control-Meta-H puts point before and the mark after the current or next DEFUN. See section 20.6.2 [Control-Meta-H], page 95.

Control-Meta-K (^R Kil Sexp)
Control-Meta-K kills the following s-expression. See section 20.6.1 [Control-Meta-K], page 94.

Control-Meta-N (^R Next List)
Control-Meta-N moves forward over one list, ignoring atoms before the first open parenthesis. See section 20.6.1 [Control-Meta-N], page 94.

Control-Meta-P (^R Previous List)
Control-Meta-P moves backward over one list, ignoring atoms reached before the first close parenthesis. See section 20.6.1 [Control-Meta-P], page 94.

Control-Meta-Q (^R Indent Sexp)
Control-Meta-Q adjusts the indentation of each of the lines in the following s-expression, but not the current line. See section 20.3 [Control-Meta-Q], page 89.

Control-Meta-T (^R Transpose Sexps)
Control-Meta-T transposes two consecutive s-expressions. See section 20.6.1 [Control-Meta-T], page 94.

Control-Meta-U (^R Backward Up List)
Control-Meta-U moves backward up one level of list structure. See section 20.6.1 [Control-Meta-U], page 94.

Files

Meta-. (^R Find Tag)
Meta-. moves to the definition of a specific function, switching files if necessary. See section 21 [Meta-.], page 101.

Meta-~ (^R Buffer Not Modified)
Meta-~ clears the flag which says that the buffer contains changes that have not been saved. See section 13.1 [Meta-~], page 55.

Control-X Control-F (Find File)
Control-X Control-F visits a file in its own buffer. See section 14 [Control-X Control-F],

page 67.

Control-X Control-Q (^R Do Not Write File)
: Control-X Control-Q tells EMACS not to offer to save this file. See section 13 [Control-X Control-Q], page 55.

Control-X Control-R (^R Read File)
: Control-X Control-R visits a file and tells EMACS not to offer to save it. See section 13.1 [Control-X Control-R], page 55.

Control-X Control-S (^R Save File)
: Control-X Control-S saves the visited file. See section 13.1 [Control-X Control-S], page 55.

Control-X Control-V (^R Visit File)
: Control-X Control-V visits a file. See section 13.1 [Control-X Control-V], page 55.

Control-X Control-W (Write File)
: Control-X Control-W saves the file, asking for names to save it under. See section 13.7 [Control-X Control-W], page 63.

File Directories

Control-X D (^R DIRED)
: Control-X D invokes the directory editor DIRED, useful for deleting many files. See section 13.6 [Control-X D], page 60.

Control-X Control-D (^R Directory Display)
: Control-X Control-D displays a subset of a directory. See section 13.1 [Control-X Control-D], page 55.

Buffers

Control-X A (^R Append to Buffer)
: Control-X A adds the text of region into another buffer. See section 9.3 [Control-X A], page 38.

Control-X B (Select Buffer)
: Control-X B is the command for switching to another buffer. See section 14 [Control-X B], page 67.

Control-X K (Kill Buffer)
: Control-X K kills a buffer. See section 14 [Control-X K], page 67.

Comments

Meta-Linefeed (^R Indent New Comment Line)
: Meta-Linefeed moves to a new line and indents it. If point had been within a comment on the old line, a new comment is started on the new line and indented under the old one. See section 20.5 [Meta-Linefeed], page 91.

Meta-; (^R Indent for Comment)
: Meta-; inserts a properly indented comment at the end of the current line, or adjusts the indentation of an existing comment. See section 20.5 [Meta-;], page 91.

Meta-N (^R Down Comment Line)
: Meta-N moves down a line and starts a comment, deleting empty comments. See section 20.5 [Meta-N], page 91.

Meta-P (^R Up Comment Line)
: Meta-P moves down a line and starts a comment, deleting empty comments. See section 20.5 [Meta-P], page 91.

Control-Meta-; (^R Kill Comment)
: Control-Meta-; kills any comment on the current line. See section 20.5 [Control-Meta-;], page 91.

Control-X ; (^R Set Comment Column)
: Control-X ; sets the column at which comments are indented, from an argument, the current column, or the previous comment. See section 20.5 [Control-X ;], page 91.

Case Conversion

Meta-C (^R Uppercase Initial)
: Meta-C makes the next word lower case with a capital initial. It moves over the word. See section 11.5 [Meta-C], page 49.

Meta-L (^R Lowercase Word)
: Meta-L moves over a word converting it to lower case. See section 11.5 [Meta-L], page 49.

Meta-U (^R Uppercase Word)
: Meta-U moves over a word converting it to upper case. See section 11.5 [Meta-U], page 49.

Control-X Control-L (^R Lowercase Region)
: Control-X Control-L converts the text of the region to lower case. See section 11.5 [Control-X Control-L], page 49.

Control-X Control-U (^R Uppercase Region)
: Control-X Control-U converts the text of the region to upper case. See section 11.5 [Control-X Control-U], page 49.

Windows

Control-Meta-V (^R Scroll Other Window)
: Control-Meta-V scrolls the other window up or down. See section 15 [Control-Meta-V], page 71.

Control-X 1 (^R One Window)
: Control-X 1 stops displaying two windows. See section 16 [Control-X 1], page 73.

Control-X 2 (^R Two Windows)
: Control-X 2 displays two windows. See section 16 [Control-X 2], page 73.

Control-X 3 (^R View Two Windows)
: Control-X 3 displays two windows but stays in the first one. See section 16 [Control-X 3], page 73.

Control-X 4 (^R Visit in Other Window)
: Control-X 4 displays two windows and selects a buffer or visits a file in the other window. See section 16 [Control-X 4], page 73.

Control-X O (^R Other Window)
: Control-X O switches from one window to the other. See section 16 [Control-X O], page 73.

Control-X ^ (^R Grow Window)
: Control-X ^ changes the allocation of screen space to the two windows. See section 16 [Control-X ^], page 73.

Narrowing

Control-X N (^R Set Bounds Region)
: Control-X N narrows the virtual boundaries to the region as it was before the command. See section 17 [Control-X N], page 77.

Control-X P (^R Set Bounds Page)
: Control-X P narrows the virtual boundaries to the current page. See section 18 [Control-X P], page 79.

Control-X W (^R Set Bounds Full)
: Control-X W widens the virtual boundaries back to the entire buffer. See section 17 [Control-X W], page 77.

Status Information

Control-X = (What Cursor Position)
: Control-X = prints information on the screen position and character position of the cursor, the size of the file, and the character after the cursor. See section 11.4 [Control-X =], page 47.

Control-X L (^R Count Lines Page)
: Control-X L prints the number of lines in the current page, and how many come before or after point. See section 18 [Control-X L], page 79.

Keyboard Macros

Control-X ((^R Start Kbd Macro)
: Control-X (begins defining a keyboard macro. See section 22.8 [Control-X (], page 124.

Control-X) (^R End Kbd Macro)
: Control-X) terminates the definition of a keyboard macro. See section 22.8 [Control-X)], page 124.

Control-X E (^R Call Last Kbd Macro)
: Control-X E executes the most recently defined keyboard macro. See section 22.8 [Control-X E], page 124.

Control-X Q (^R Kbd Macro Query)
: Control-X Q in a keyboard macro can ask the user whether to continue or allow him to do some editing before continuing with the keyboard macro. See section 22.8 [Control-X Q], page 124.

Minibuffer

Control-% (^R Replace String)
: Control-% invokes a minibuffer containing a call to Replace String. You fill in the arguments. See section 19 [Control-%], page 83.

Meta-Altmode (^R Execute Minibuffer)
: Meta-Altmode invokes an empty minibuffer which you can fill in with a TECO program to be executed. See section 23 [Meta-Altmode], page 127.

Meta-% (^R Query Replace)
: Meta-% invokes a minibuffer containing a call to Query Replace. You fill in the arguments. See section 19 [Meta-%], page 83.

Control-X Altmode (^R Re-execute Minibuffer)
: Control-X Altmode re-executes a TECO program previously executed in the minibuffer. It can also re-execute an extended command. See section 23 [Control-X Altmode], page 127.

Catalog of Libraries

Libraries Used Explicitly

These are libraries which you must load with M-X Load Library♦<libname><cr> to use. If no cross-reference is given, the only documentation for the library is the self-documentation contained in it. Use M-X List Library♦<libname><cr> to print a brief description of each function in the library. For more detailed information, load the library and use M-X Describe on individual functions.

ABSTR	contains commands for making documentation files: wall charts, and abstracts of libraries. See the file INFO;CONV >, node Top.
AUTO-S	is an alternate implementation of Auto Save mode. It has some features which the standard version lacks, and lacks some which the standard version has.
BABYL	is a subsystem for reading, sending and editing mail. See the file INFO;BABYL >, node Top.
BCPL	implements BCPL mode.
BLISS	implements BLISS mode.
CHESS	implements commands for editing pictures of chessboards.
COLUMNS	implements commands for converting single-column text into double-column text and vice versa.
DELIM	implements commands for moving over balanced groupings of various kinds of parentheses. There are a pair of commands for square brackets, a pair for angle brackets, etc.
DOCLSP	prints documentation from the MacLisp manual on a specified Lisp function.
DOCOND	is a macro processor and conditionalizer for text files, useful for maintaining multiple versions of documents with one source.
EAKMACS	EAK's personal library.
FORTRAN	implements FORTRAN mode.
HAZ1510	redefines commands to be convenient on Hazeltine 1510 terminals.
INFO	peruses tree-structured documentation files.
IVORY	is EAK and ECC's own generator for EMACS libraries, which uses a slightly different input format. The libraries EAKMACS, IVORY, MKDUMP, TMACS and WORDAB, and all of BABYL, are generated with IVORY.

JOURNAL implements journal files. See section 24.4 [Journals], page 133.

LEDIT is the EMACS side of the EMACS-to-MacLisp interface. See the file INFO;LEDIT >.

LISPT is the EMACS side of another EMACS-to-MacLisp interface. See the file INFO;LISPT >.

LUNAR is Moon's personal library.

MACCNV does part of the work of converting MACRO-10 code to MIDAS code.

MAZLIB is a game for solving mazes. It's fun to play.

MKDUMP aids in dumping your own customized environment.

MODLIN implements a fancier mode line display.

MQREPL works with TAGS to perform several Query Replaces on each of the files in a tag table.

NEWS is for reading the latest AP or New York Times news summary.

OUTLIN implements Outline mode, for editing outlines.

PAGE defines commands for viewing only one page of the file at a time. See section 18.1 [PAGE], page 80.

PASCAL implements PASCAL mode. See the file INFO;EPASC >.

PHRASE has commands for moving over and killing phrases of text.

PICTURE contains Edit Picture, the command for editing text pictures. See section 26 [PICTURE], page 151.

PL1 implements PL1 mode. See the file INFO;EPL1 >.

PURIFY generates libraries from EMACS source files, and contains other functions useful for editing the source files. See the file INFO;CONV >.

QSEND sends a message to another logged-in user, like :QSEND.

RENUM renumbers figures, equations, theorems or chapters.

RMAIL is for reading, editing and sending mail.

RUNOFF is for text-justified documents divided into separate source files. It rejustifies the files which have changed, then runs :@ to print only the pages which have changed.

SCRLIN contains alternative definitions of C-N and C-P which move by screen lines instead of by real lines.

SLOWLY redefines commands and options to suit slow terminals.

TDEBUG is a debugger for TECO programs. It displays the buffer in one window and the program in the other, while stepping by lines or setting breakpoints. See the file INFO;TDEBUG >, node Top.

TEX implements TEX mode.

TIME causes the current time of day to be displayed in the mode line.

TMACS contains miscellaneous useful functions

VT100 defines the arrow keys and numeric keypad of the VT-100 terminal to move the cursor and supply numeric arguments.

VT52 defines the numeric keypad of the VT-52 terminal to supply numeric arguments.

Automatically Loaded Libraries

These are libraries which the user need not know about to use.

AUX implements several commands described in the manual as part of the standard EMACS. Loaded automatically when needed.

BABYLV is an auxiliary library for BABYL.

BARE contains the definitions of all built-in functions. These definitions are not needed for executing the built-in functions, only so that Help can describe them properly. Loaded automatically by documentation commands when needed. See section 5.2 [BARE], page 21.

DIRED implements the commands for editing and listing directories. Loaded automatically when needed. See section 13.6 [DIRED], page 60.

EINIT is used in building and dumping EMACS. See the file INFO;CONV >.

EMACS is the main body of standard EMACS. Always loaded.

GRIND implements C-M-G. Loaded automatically when needed. See section 20.7 [Grinding], page 96.

KBDMAC implements keyboard macros. Loaded automatically when needed. See section 22.8 [Keyboard Macros], page 124.

KEYSET is an auxiliary file for BABYL.

SORT implements the sorting commands. Loaded automatically when needed.

SYMENU is an auxiliary file for BABYL.

TAGS implements the TAGS package. See section 21 [TAGS], page 101.

WORDAB implements Word Abbrev mode, loaded automatically when needed. See section 25 [WORDAB], page 141.

Index of Variables

An option is a variable whose value Edit Options offers for editing. A hook variable is a variable which is normally not defined, but which you can define if you wish for customization. Most hook variables require TECO programs as their values.

The default value of the variable is given in parentheses after its name. If no value is given, the default value is zero. If the word "nonexistent" appears, then the variable does not exist unless you create it.

Abort Resumption Message
This is the message to be printed by C-] to tell you how to resume the aborted command. If this variable is zero, there is no way to resume, so C-] asks for confirmation. See section 24.1 [Quitting], page 129.

Additional Abbrev Expanders (nonexistent)
If this variable exists when Word Abbrev Mode is turned on, it is string of characters which should terminate and expand an abbrev, in addition to the punctuation characters which normally do so. See also WORDAB Ins Chars.

Atom Word Mode
The minor mode Atom Word mode is on if this variable is nonzero. See section 22.1 [Atom Word Mode], page 111.

Auto Directory Display
If this is nonzero, certain file operations automatically display the file directory. See section 13.1 [Auto Directory Display], page 55.

Auto Fill Mode The minor mode Auto Fill mode is on if this variable is nonzero. See section 11.4 [Auto Fill Mode], page 47.

Auto Push Point Notification
The value of this variable is the string printed in the echo area by some commands to notify you that the mark has been set to the old location of point. See section 10 [Auto Push Point Notification], page 41.

Auto Push Point Option (500)
Searches set the mark if they move at least this many characters. See section 10 [Auto Push Point Option], page 41.

Auto Save Default
The minor mode Auto Save mode is on by default for newly visited files if this variable is nonzero. See section 13.3 [Auto Save Default], page 57.

Auto Save Filenames (DSK:<working directory>;_^RSV >)
These are the filenames used for auto saving if the visited filenames are not used. See section 13.3 [Auto Save Filenames], page 57.

Auto Save Interval (500)
This is the number of characters between auto saves. See section 13.3 [Auto Save Interval],

page 57.

Auto Save Max (2)
: This is the maximum number of auto saves to keep. See section 13.3 [Auto Save Max], page 57.

Auto Save Visited File
: If this is nonzero, auto saving saves as the visited filenames. If this is zero, auto saving saves as the names which are the value of Auto Save Filenames (q.v.). See section 13.3 [Auto Save Visited File], page 57.

Autoarg Mode
: When Autoarg Mode is nonzero, numeric arguments can be specified just by typing the digits. See section 4 [Autoarg Mode], page 17.

Buffer Creation Hook (nonexistent)
: If this variable exists, its value should be a TECO program to be executed whenever a newly created buffer is selected for the first time. See section 14 [Buffer Creation Hook], page 67.

Buffer Deselection Hook (nonexistent)
: If this variable exists, its value should be a TECO program to be executed whenever a buffer is about to be deselected. The difference between this and Buffer Selection Hook is that, while both are executed (if they exist) when you switch buffers, this is executed before the switch, and Buffer Selection Hook is executed after the switch. See section 14 [Buffer Deselection Hook], page 67.

Buffer Selection Hook (nonexistent)
: If this variable exists, its value should be a TECO program to be executed whenever a buffer is selected. See section 14 [Buffer Selection Hook], page 67.

Case Replace (1)
: When Case Replace is nonzero, Replace String and Query Replace attempt to preserve case when they replace. See section 19 [Case Replace], page 83.

Comment Begin
: This is the string used to start comments. If the value is zero, semicolon is used. See section 20.5 [Comment Begin], page 91.

Comment Column
: This is the column at which comments are aligned. See section 20.5 [Comment Column], page 91.

Comment End
: This is the string which is used to end comments. It is often empty for languages in which comments end at the end of the line. See section 20.5 [Comment End], page 91.

Comment Rounding (/8+1*8)
: This is the TECO program used to decide what column to start a comment in when the text of the line goes past the comment column. The argument to the program is the column at which text ends. See section 20.5 [Comment Rounding], page 91.

Comment Start
: This is the string used for recognizing existing comments, and for starting new ones if Comment Begin is zero. If Comment Start is zero, semicolon is used. See section 20.5 [Comment Start], page 91.

Compile Command (nonexistent)

If this variable exists, its value should be a TECO program to be used by the M-X Compile command to compile the file. See section 20.2 [Compile Command], page 88.

Default Major Mode (Fundamental)
This is the major mode in which new buffers are created. If it is the null string, new buffers are created in the same mode as the previously selected buffer. See section 14 [Default Major Mode], page 67.

Directory Lister (& Subset Directory Listing)
This is the TECO program used for listing a directory for C-X C-D and the Auto Directory Display option. The default value is the definition of the function & Subset Directory Listing. Another useful value is the definition of the function & Rotated Directory Listing.

Display Matching Paren (-1)
This variable controls automatic display of the matching open parenthesis when a close parenthesis is inserted. See section 20.4 [Display Matching Paren], page 90.

EMACS Version This variable's value is the EMACS version number.

Exit Hook (nonexistent)
If this variable exists, its value should be a TECO program to be executed whenever EMACS is exited. The subroutine & Exit EMACS is responsible for executing it. See section 6.3 [Exit Hook], page 26.

Fill Column (79) The value of Fill Column is the width used for filling text. See section 11.4 [Fill Column], page 47.

Fill Extra Space List (.?!)
The characters in this string are the ones which ought to be followed by two spaces when text is filled. See section 11.4 [Fill Extra Space List], page 47.

Fill Prefix The value of this variable is the prefix expected on every line of text before filling and placed at the front of every line after filling. It is usually empty, for filling nonindented text. See section 11.4 [Fill Prefix], page 47.

Find File Inhibit Write
If this variable is nonzero, then C-X C-F visits files in read-only (C-X C-R) fashion. Normally, C-X C-F visits files as if C-X C-V were being used. See section 14 [Find File Inhibit Write], page 67.

Indent Tabs Mode (-1)
If Indent Tabs Mode is nonzero, then tab characters are used by the indent commands. Otherwise, only spaces are used. See section 11.3 [Indent-Tabs Mode], page 46.

Inhibit Write If Inhibit Write is nonzero, then there will be no offer to save the visited file if another file is visited in the same buffer. C-X C-R sets this variable nonzero. See section 13.1 [Inhibit Write], page 55.

<libname> Setup Hook (nonexistent)
If this variable exists, its value should be a TECO program to be executed when the library <libname> is loaded. The library's Setup function is responsible for doing this. If the library has no Setup function, it will not handle a setup hook either. See section 22.2

[Libraries], page 112.

<mode> ..D (nonexistent)
: This variable is used by the major mode <mode> to record the syntax table for that mode. It is created by the first use of the mode, and if you supply your value, that value will be accepted instead. For example, Text mode uses Text ..D. See section 22.4 [Syntax], page 115.

<mode> Mode Hook (nonexistent)
: If this variable exists, its value should be a TECO program to be executed when the major mode <mode> is entered. For example, Text Mode Hook is executed when Text mode is entered. See section 20 [Major Modes], page 87.

Next Screen Context Lines (nonexistent)
: If this variable exists, its value should be the number of lines of overlap between one screenful and the next, when scrolling by screens with C-V and M-V. See section 15 [Next Screen Context Lines], page 71.

Only Global Abbrevs (nonexistent)
: If this variable exists and its value is nonzero, then Word Abbrev Mode assumes that you are not using any mode-specific abbrevs. See section 25.2.1 [Only Global Abbrevs], page 145.

Overwrite Mode If this is nonzero, the minor mode Overwrite mode is in effect. See section 22.1 [Overwrite Mode], page 111.

Page Delimiter (↑L)
: This is the TECO search string used to recognize page boundaries. See section 18 [Page Delimiter], page 79.

PAGE Flush CRLF (0)
: If this is nonzero, the PAGE library expects every page to start with a blank line which is not considered part of the contents of the page. See section 18.1 [PAGE Flush CRLF], page 80.

Paragraph Delimiter (.↑O ↑O ↑O'↑O@)
: This is the TECO search string used to recognize beginnings of paragraphs. See section 11.2 [Paragraph Delimiter], page 44.

Permit Unmatched Paren (-1)
: Controls whether the bell is run if you insert an unmatched close parenthesis. See section 20.4 [Permit Unmatched Paren], page 90.

Read Line Delay This is the amount of time, in 30'ths of a second, which EMACS should wait after starting to read a line of input, before it prompts and starts echoing the input.

Region Query Size (5000)
: Many commands which act on the region require confirmation if the region contains more than this many characters.

Return from Superior Hook (nonexistent)
: If this variable exists, its value should be a TECO program to be executed whenever EMACS is resumed after being exited. See section 6.3 [Return from Superior Hook],

page 26.

Set Mode Line Hook
This is a hook which is executed every time the mode line is recomputed. It can insert text in the buffer to put it in the mode line after the minor modes. See section 1.1 [Set Mode Line Hook], page 6.

Space Indent Flag
If this flag is nonzero, then Auto Fill indents the new lines which it creates, by performing a Tab. Most major modes for programming languages set this nonzero. See section 11.4 [Space Indent Flag], page 47.

Tab Stop Definitions (a string)
The value of Tab Stop Definitions is a string defining the tab stops to be used by the command M-I (^R Tab to Tab Stop). See section 11.3 [Tab Stop Definitions], page 46.

Tags Find File (nonexistent)
If this variable exists and is not zero, TAGS uses C-X C-F to switch files. Otherwise, TAGS uses C-X C-V. See section 21 [TAGS], page 101.

Temp File FN2 List (MEMO↑OXGP↑O ...) This is a TECO search string which recognizes the filenames which indicate that the file is probably temporary. See section 13.5 [Temp File FN2 List], page 59.

Underline Begin (nonexistent)
If this variable exists, its value should be the character or string to use to begin underlines for the M-_ command. See section 11.7 [Underline Begin], page 51.

Underline End (nonexistent)
If this variable exists, its value should be the character or string to use to end underlines for the M-_ command. See section 11.7 [Underline End], page 51.

Visit File Hook (nonexistent)
If this variable exists, its value should be a TECO program to be executed whenever a file is visited. See section 13.1 [Visit File Hook], page 55.

WORDAB Ins Chars (nonexistent)
If this variable exists when Word Abbrev Mode is turned on, it should be a string containing precisely those characters which should terminate and expand an abbrev. This variable overrides Additional Abbrev Expanders (q.v.). See section 25.2.1 [WORDAB], page 145.

Non-Control Non-Meta Characters:

```
Backspace ↑R Backward Character
Tab       ↑R Indent According to Mode
Linefeed  ↑R Indent New Line
Return    ↑R CRLF
Altmode   ↑R Prefix Meta
Rubout    ↑R Backward Delete Character
```

Control Characters:

```
Alpha     ↑R Complement SAIL Mode
Altmode   ↑R Exit
Space     ↑R Set/Pop Mark
%    ..   ↑R Replace String
-    ..   ↑R Negative Argument
0 thru 9  ↑R Argument Digit
;    ..   ↑R Indent for Comment
<    ..   ↑R Mark Beginning
=    ..   What Cursor Position
>    ..   ↑R Mark End
@    ..   ↑R Set/Pop Mark
A    ..   ↑R Beginning of Line
B    ..   ↑R Backward Character
C    ..   ↑R Prefix Control-Meta
D    ..   ↑R Delete Character
E    ..   ↑R End of Line
F    ..   ↑R Forward Character
G    ..   ↑R Quit
H    ..   ↑R Backward Character
I    ..   ↑R Indent According to Mode
J    ..   ↑R Indent New Line
K    ..   ↑R Kill Line
L    ..   ↑R New Window
M    ..   ↑R Self Insert for Formatting Character
N    ..   ↑R Down Real Line
O    ..   ↑R Open Line
P    ..   ↑R Up Real Line
Q    ..   ↑R Quoted Insert
R    ..   ↑R Reverse Search
S    ..   ↑R Incremental Search
T    ..   ↑R Transpose Characters
U    ..   ↑R Universal Argument
V    ..   ↑R Next Screen
W    ..   ↑R Kill Region
X    ..   is a prefix character.  See below.
Y    ..   ↑R Un-kill
Z    ..   ↑R Return to Superior
\    ..   ↑R Prefix Meta
]    ..   Abort Recursive Edit
↑    ..   ↑R Prefix Control
Rubout    ↑R Backward Delete Hacking Tabs
```

Meta Characters:

```
Linefeed    ↑R Indent New Comment Line
Return      ↑R Back to Indentation
Altmode     ↑R Execute Minibuffer
#     ..    ↑R Change Font Word
%     ..    ↑R Query Replace
'     ..    ↑R Upcase Digit
(     ..    ↑R Make ()
)     ..    ↑R Move Over )
-     ..    ↑R Negative Argument
.     ..    ↑R Find Tag
/     ..    ↑R Describe
0 thru 9    ↑R Argument Digit
;     ..    ↑R Indent for Comment
<     ..    ↑R Goto Beginning
=     ..    ↑R Count Lines Region
>     ..    ↑R Goto End
?     ..    ↑R Describe
@     ..    ↑R Mark Word
A     ..    ↑R Backward Sentence
B     ..    ↑R Backward Word
C     ..    ↑R Uppercase Initial
D     ..    ↑R Kill Word
E     ..    ↑R Forward Sentence
F     ..    ↑R Forward Word
G     ..    ↑R Fill Region
H     ..    ↑R Mark Paragraph
I     ..    ↑R Tab to Tab Stop
J     ..    ↑R Indent New Comment Line
K     ..    ↑R Kill Sentence
L     ..    ↑R Lowercase Word
M     ..    ↑R Back to Indentation
N     ..    ↑R Down Comment Line
P     ..    ↑R Up Comment Line
Q     ..    ↑R Fill Paragraph
R     ..    ↑R Move to Screen Edge
S     ..    ↑R Center Line
T     ..    ↑R Transpose Words
U     ..    ↑R Uppercase Word
V     ..    ↑R Previous Screen
W     ..    ↑R Copy Region
X     ..    ↑R Extended Command
Y     ..    ↑R Un-kill Pop
[     ..    ↑R Backward Paragraph
\     ..    ↑R Delete Horizontal Space
]     ..    ↑R Forward Paragraph
^     ..    ↑R Delete Indentation
_     ..    ↑R Underline Word
~     ..    ↑R Buffer Not Modified
Rubout      ↑R Backward Kill Word
```

Control-Meta Characters:

Backspace	↑R Mark Defun
Tab	↑R Indent for LISP
Linefeed	↑R Indent New Comment Line
Return	↑R Back to Indentation
(..	↑R Backward Up List
) ..	↑R Forward Up List
- ..	↑R Negative Argument
0 thru 9	↑R Argument Digit
; ..	↑R Kill Comment
? ..	↑R Documentation
@ ..	↑R Mark Sexp
A ..	↑R Beginning of DEFUN
B ..	↑R Backward Sexp
C ..	↑R Exit
D ..	↑R Down List
E ..	↑R End of DEFUN
F ..	↑R Forward Sexp
G ..	↑R Format Code
H ..	↑R Mark Defun
I ..	↑R Indent for LISP
J ..	↑R Indent New Comment Line
K ..	↑R Kill Sexp
M ..	↑R Back to Indentation
N ..	↑R Forward List
O ..	↑R Split Line
P ..	↑R Backward List
Q ..	↑R Indent SEXP
R ..	↑R Reposition Window
T ..	↑R Transpose Sexps
U ..	↑R Backward Up List
V ..	↑R Scroll Other Window
W ..	↑R Append Next Kill
X ..	↑R Instant Extended Command
[..	↑R Beginning of DEFUN
\ ..	↑R Indent Region
] ..	↑R End of DEFUN
↑ ..	↑R Delete Indentation
Rubout	↑R Backward Kill Sexp

Control-X is an escape prefix command with these subcommands:

```
↑X ↑B      List Buffers
↑X ↑C      ↑R Return to Superior
↑X ↑D      ↑R Directory Display
↑X ↑F      Find File
↑X Tab     ↑R Indent Rigidly
↑X ↑L      ↑R Lowercase Region
↑X ↑N      ↑R Set Goal Column
↑X ↑O      ↑R Delete Blank Lines
↑X ↑P      ↑R Mark Page
↑X ↑Q      ↑R Do Not Write File
↑X ↑R      ↑R Read File
↑X ↑S      ↑R Save File
↑X ↑T      ↑R Transpose Lines
↑X ↑U      ↑R Uppercase Region
↑X ↑V      ↑R Visit File
↑X ↑W      Write File
↑X ↑X      ↑R Exchange Point and Mark
↑X Altmode ↑R Re-execute Minibuffer
↑X #       ↑R Change Font Region
↑X (       ↑R Start Kbd Macro
↑X .       ↑R Set Fill Prefix
↑X 1       ↑R One Window
↑X 2       ↑R Two Windows
↑X 3       ↑R View Two Windows
↑X 4       ↑R Visit in Other Window
↑X ;       ↑R Set Comment Column
↑X =       What Cursor Position
↑X A       ↑R Append to Buffer
↑X B       Select Buffer
↑X D       ↑R Dired
↑X F       ↑R Set Fill Column
↑X G       ↑R Get Q-reg
↑X H       ↑R Mark Whole Buffer
↑X I       ↑R Info
↑X K       Kill Buffer
↑X L       ↑R Count Lines Page
↑X M       Mail
↑X N       ↑R Set Bounds Region
↑X O       ↑R Other Window
↑X P       ↑R Set Bounds Page
↑X R       Read Mail
↑X T       ↑R Transpose Regions
↑X W       ↑R Set Bounds Full
↑X X       ↑R Put Q-reg
↑X [       ↑R Previous Page
↑X ]       ↑R Next Page
↑X ↑       ↑R Grow Window
↑X ←       ↑R Underline Region
↑X Rubout  ↑R Backward Kill Sentence
```

Index

! 83

& Exit EMACS 27
& Alter..D 117

. 83

<cr> 9

C-C 10, 128
C-C C-Y 128
C-M-C 26, 60, 114

Abbrev definition files 144, 147
Abbrev definition lists 141, 144, 146, 147
Abbrev profusion 147
Abbrevs 141
Abbrevs in multitude 147
Abort Recursive Edit 114, 129
Aborting 129
Accumulator 97
Additional Abbrev Expanders 146
Address 97
Altmode 9, 10, 20, 83, 84, 117, 126, 127
Append to File 38, 63
Apropos 29
ASCII 9
Atom word mode 6, 44, 111
Auto Directory Display 59
Auto fill 93
Auto fill mode 6, 46, 47, 92, 111, 112, 118, 120
Auto Push Point Notification 32
Auto Push Point Option 32
Auto Save Default 57, 111
Auto Save Filenames 58
Auto Save Interval 59
Auto Save Max 58
Auto save mode 6, 57, 111
Auto Save Visited File 58
Autoarg mode 18, 157
Autoloading 113

BARE library 23
Bit prefix characters 156
Blank lines 15, 30, 45, 92, 93
Buffer Creation Hook 59
Buffers 6, 67, 74, 102, 118

C- 9
C-; 91
C-< 32
C-> 32
C-@ 31

C-A 13, 45, 89
C-B 13, 85
C-D 14, 19, 35
C-E 13, 45, 89
C-F 13
C-G 19, 26, 42, 55, 128, 129, 133
C-K 14, 35, 45
C-L 13, 71, 126, 161
C-M-(95
C-M-) 95
C-M-; 91
C-M-@ 32, 95
C-M-A 95, 97
C-M-B 94
C-M-D 95, 97
C-M-E 95, 97
C-M-F 94
C-M-G 97, 98
C-M-H 32, 95
C-M-K 35, 95
C-M-L 79
C-M-M 46, 89
C-M-N 94, 97
C-M-P 94, 97
C-M-Q 97
C-M-R 72
C-M-Rubout 35, 95
C-M-Space 143, 148
C-M-T 95
C-M-Tab 96
C-M-U 95
C-M-V 73
C-M-W 37
C-M-X 20
C-M-[95
C-M-\ 47, 97
C-M-] 95
C-M-t 89, 96
C-N 13, 85
C-O 15, 85, 89
C-P 13
C-Q 13, 85, 143, 147
C-R 41, 81, 83, 126, 159
C-Rubout 93
C-S 41, 81, 159
C-Space 31
C-T 13, 53
C-U 17, 19, 55
C-U C-@ 45
C-V 71
C-W 35, 45, 79, 83
C-X 10, 85, 118, 121
C-X # 50
C-X (142, 145, 146, 148
C-X) 142, 146, 148
C-X 1 73
C-X 2 73
C-X 3 74
C-X 4 74

C-X C-C 26
C-X A 38
C-X Altmode 21, 84, 128
C-X B 67
C-X C-A 142, 145, 148
C-X C-B 68
C-X C-D 59
C-X C-F 67
C-X C-H 142, 148
C-X C-L 50
C-X C-O 15, 35
C-X C-P 32, 79, 80
C-X C-S 14, 55, 58, 68
C-X C-T 54
C-X C-U 31, 50
C-X C-V 14, 55
C-X C-W 58
C-X C-X 31
C-X D 62
C-X E 125
C-X F 48, 111
C-X G 39
C-X H 32
C-X J 81
C-X K 68
C-X L 80
C-X N 77, 83
C-X O 73
C-X P 79, 81
C-X Q 126, 159
C-X Rubout 35, 45, 53
C-X T 54
C-X Tab 47
C-X U 148
C-X W 77, 79, 81, 83
C-X X 39
C-X ← 51
C-X (125
C-X) 125
C-X . 48
C-X ; 92
C-X = 49
C-X [79, 81
C-X] 79, 81
C-X ↑ 74, 128
C-Y 37
C-\ 127
C-] 60, 84, 114, 117, 129
C-↑ 10
C-← 29
Capitalization 143
Case conversion 49, 54, 83, 141, 157
Case Replace 83
Centering 48
Character Search 42
Character set 9
Clean Directory 60
Clear screen 71
Comma 83

Command completion 20, 23
Commands 19
Comment Begin 93
Comment Column 92, 119
Comment End 93
Comment S 93, 119
Commen 81, 87, 91, 93, 94, 97, 116
Compare .tories 64
Compile C. nand 88
Confirmation 50, 56, 57, 61, 128, 142
Continuation line 13
Control 9, 134
Control-Meta 94
Copy File 63
Create File 56
CRLF 13
Cursor 5
Customization 22, 118, 120, 145

Default Separator 160
Define Word Abbrevs 146
Defining abbrevs 142, 144
Defuns 32, 95
Delete File 63
Deletion 13, 35, 53, 84, 132
Describe 29, 115
Directory 59, 62, 64
Directory Lister 59
DIRED 59, 60, 11?
Display Matching F. 90
Documentation 30
Down Picture Mover t 151
Dumped environments 147

Echo area 5, 19, 49, 79, 142
Edit Indented Text 46
Edit key 156
Edit Options 26, 114
Edit Picture 26, 151
Edit Some Word Abbrevs 147
Edit Syntax Table 117
Edit Tab Stops 43, 47
Edit Word Abbrevs 144
Editor Type 6
Error message 5
Escape key 155
EVARS files 112, 118, 120, 145, 146
Exit Hook 27
Exiting 26
Expander characters 146
Expanding abbrevs 143
Extended commands 19

FAIL 97
File deletion 59, 60
File directory 59
File Versions Kept 60
Files 6, 14, 55, 57, 63, 74, 118
Fill Column 48, 111, 114

Fill Prefix 45, 48
Filling 47
Find File 102
Flush Lines 84
Fonts 50
Formatting 46, 96
Formfeed 79
FS Flags 117
Functions 19

Global abbrevs 141, 145, 146
Grinding 96

Help 23, 29, 134
Home Directory 120, 133
Hooks 148
How Many 84

Incremental abbrev definition files 147
Indent Tabs Mode 47, 112
Indentation 46, 89, 91, 96
Init file 157
Init files 120, 145, 146
Insert File 63
Insert Page Directory 81
Insert Some Word Abbrevs 147
Insert Word Abbrevs 146
Insertion 13, 63, 79
INTER 93

Join pages 81
Journal files 133

Keep Lines 62, 84
Keyboard macros 6, 125
Kill All Word Abbrevs 146
Kill Buffer 68
Kill Libraries 113, 131
Kill ring 37, 131
Kill Some Buffers 68, 105, 131
Killing 35, 37, 43, 45, 53, 95, 132
Killing abbrevs 143, 144, 146

Labels 97
Large numbers of abbrevs 147
LEDIT 6, 93
Left Picture Movement 151
Libraries 80, 112, 141, 151
Linefeed 46, 87, 89, 96
Lines 35, 153
Lisp 89, 111
Lisp mode 93
List Buffers 68
List Files 59, 63
List Library 113
List Loaded Libraries 113
List Some Word Abbrevs 147
List Variables 115
List Word Abbrevs 144, 146

Lists 32, 94, 116
Load Library 112
Loading 112
Local modes lists 112
Local variables 67, 115, 118

M- 9
M-" 98
M-# 50
M-% 84, 127
M-' 54, 98, 143, 148
M-(95
M-) 95
M-- M-C 54
M-- M-L 54
M-- M-U 54
M-. 103
M-; 91
M-< 13
M-> 13
M-@ 32, 44
M-A 45
M-Altmode 127
M-B 43
M-C 49
M-D 35, 43
M-E 45
M-F 43
M-G 48
M-H 32, 45, 48
M-I 47
M-J 92
M-K 35, 45
M-L 49
M-Linefeed 92
M-M 46, 89
M-N 92
M-O 159
M-P 92
M-Q 48
M-R 72
M-Rubout 35, 43, 53
M-S 48
M-T 44
M-U 49
M-V 71
M-W 37
M-X 19, 21
M-Y 37
M-[45, 97
M-\ 35, 46, 89
M-] 45, 97
M-^ 35, 46, 89, 96
M-← 51
M.I. 115
Macro-10 97
Macsyma 97
Macsyma mode 98
MAIL.T 6

Major modes 6, 87, 92, 117, 118, 141
Make Space 131
Make These Characters Expand 146
Make Word Abbrev 145
Many abbrevs 147
Mark 31, 36, 37, 44, 45, 63, 79, 95
Matching 90
Meta 9, 43, 134, 155, 156
Metizer 9, 127, 156
MIDAS 88, 89, 97
MIDAS mode 97
Minibuffer 21, 26, 62, 71, 84, 127
Minibuffer Separator 160
Minor modes 6, 47, 111, 141
MM 19, 39
Mode abbrevs 141, 145, 146
Mode line 6, 26, 67, 80, 87, 111, 127
More line 6
Motion 43, 45, 79, 94, 95
Moving text 37
Muddle mode 93

Name Kbd Macro 125
Narrowing 49, 77, 79, 83
Numeric argument 142
Numeric arguments 17, 20, 22, 36, 38, 44, 47, 48, 49, 50, 51, 57, 59, 71, 73, 74, 83, 92, 96, 111, 114, 128, 156, 157

Occur 84
Only Global Abbrevs 146
Options 114
Overwrite mode 6, 111

PAGE 80
Page Delimiter 45, 80
PAGE Flush CR.F 81
Pages 32, 45, 79, 80, 153
Paragraph Delimiter 45, 98
Paragraphs 32, 44, 48, 93, 97, 153
Parentheses 43, 90
Permit Unmatched Paren 90
Pictures 151
Prefix characters 10, 118, 121
Prepend to File 38, 63
Preventing abbrev expansion 143
Printing terminal 161
Prompting 5, 19
PURIFY library 98

Q-registers 39
Query Replace 83, 126, 127
Quick List Some Word Abbrevs 147
Quit 133
Quitting 42, 129
Quoting 13, 147

R 50, 51
Read Incremental Word Abbrev File 147
Read Word Abbrev File 144, 147
Readable Word Abbrev Files 144

Reap File 60
Recursive Editing Level 26, 60, 68, 117, 129, 134, 144
Redefining abbrevs 142
Redefining commands 118, 120
Region 31, 36, 37, 45, 50, 51, 63, 77, 79, 95, 97, 142, 153
Rename Buffer 68
Rename File 63
Replace String 83
Replacement 83, 84
Replay Journal File 133
Restarting 131
Return 9, 13, 55
Revert File 57, 58
Right Picture Movement 151
RMAIL 6
Rubout 9, 13, 14, 19, 35, 53, 83, 87, 93, 111, 126
Run Library 113

S-expressions 94, 116
SAIL characters 155
Save All Files 68
Saving 55, 57
Saving abbrevs 144
Screen 5, 71
Scrolling 71, 73
Searching 41, 81, 83
Select Buffer 67
Sentences 44, 53
Set Key 146
Set Variable 114
Set Visited Filename 63
Short Display Size 159
Slow Search Lines 159
Slow Search Separator 160
SLOWLY Maximum Speed 160
Sort Lines 153
Sort Pages 153
Sort Paragraphs 153
Sorting 153
Space 9, 20, 47, 83, 126
Space Indent Flag 46, 49
Split pages 81
SRCCOM 56, 62
Start Journal File 133
String arguments 20, 22, 59, 114
String Search 42
Submode 6
Subroutines 23
Syntax table 43, 44, 90, 93, 95, 115

Tab 43, 46, 87, 89, 93, 96
Tab Stop Definitions 47
Tabify 47
Tags 74, 113
Tags Find File 102
Tags Search 105
TECO 19, 21, 39, 45, 49, 63, 71, 85, 88, 89, 90, 113, 115, 117, 120, 125, 131
TECO default filenames 63
TECO mode 98

TECO search string 60, 85, 104
Temp File FN2 List 60
Terminal type 131
Text justifiers 50, 51
Text mode 43, 90
Text Mode Hook 90
TJ6 50
Toggling 111
Too many! 147
Top Level 129, 131
Transposition 44, 53, 95
Two window mode 73
Types 53, 54

Underline Begin 51
Underline End 51
Underlining 51
Undo 37, 50, 57, 131, 132
Unexpanding abbrevs 143
Untabify 47
Up Picture Movement 151
URK 131
Usage count 144
User Name 120

Variables 114, 120
View Buffer 72
View Directory 59
View File 63
View Kbd Macro 126
View Page Directory 81
Visit File 63
Visit Tag Table 102
Visiting 14, 55, 57, 67, 74

W2 73
What Cursor Position 49
What Page 79
Windows 73
Word Abbrev Apropos 147
Word Abbrev Hook 148
Word abbrev mode 6, 111, 141, 142, 146
Word abbrev profusion 147
Word Search 42
WORDAB 141
WORDAB DEFNS 144
WORDAB Ins Chars 146
WORDAB Setup Hook 148
Words 32, 43, 49, 50, 51, 53, 54, 116
Write File 58, 63
Write Incremental Word Abbrev File 147
Write Word Abbrev File 144, 147

↑ 83
↑R Abbrev Expand Only 143
↑R Add Global Word Abbrev 142, 145, 146
↑R Add Mode Word Abbrev 147, 145
↑R Edit Quietly 159
↑R Inverse Add Global Word Abbrev 142, 145, 146

↑R Inverse Add Mode Word Abbrev 142, 145
↑R Kill Global Word Abbrev 146
↑R Kill Mode Word Abbrev 146
↑R Set Screen Size 159
↑R Slow Display I-Search 159
↑R Slow Reverse Display I-search 159
↑R Word Abbrev Prefix Mark 143
↑R Append Next Kill 37
↑R Append to Buffer 38
↑R Back to Indentation 46
↑R Backward Character 13
↑R Backward Delete Character 13, 35, 53, 93
↑R Backward Delete Hacking Tabs 93
↑R Backward Kill Sentence 35, 45, 53
↑R Backward Kill Sexp 35, 95
↑R Backward Kill Word 35, 43, 53
↑R Backward List 94
↑R Backward Paragraph 45, 97
↑R Backward Sentence 45
↑R Backward Sexp 94
↑R Backward TECO Conditional 98
↑R Backward Up List 95
↑R Backward Word 43
↑R Beginning of Defun 95
↑R Beginning of Line 13
↑R Center Line 48
↑R Change Font Region 50
↑R Change Font Word 50
↑R Copy Region 37
↑R Count Lines Page 80
↑R CRLF 13, 87
↑R Delete Blank Lines 15, 35
↑R Delete Character 35
↑R Delete Horizontal Space 35, 46, 89
↑R Delete Indentation 35, 46, 89, 96
↑R Directory Display 59
↑R DIRED 62
↑R Down Comment Line 92
↑R Down List 95
↑R Down Real Line 13
↑R End Kbd Macro 125
↑R End of Defun 95
↑R End of Line 13
↑R Exchange Point and Mark 31
↑R Execute Kbd Macro 125
↑R Execute Minibuffer 127
↑R Exit 26, 114
↑R Fill Paragraph 48
↑R Fill Region 48
↑R Find File 67
↑R Format Code 97
↑R Forward Character 13
↑R Forward List 94
↑R Forward Paragraph 45, 97
↑R Forward Sentence 45
↑R Forward Sexp 94
↑R Forward TECO Conditional 98
↑R Forward Up List 95
↑R Forward Word 43

†R Get Q-reg 39
†R Go to AC Field 97
†R Go to Address Field 97
†R Go to Next Label 97
†R Go to Previous Label 97
†R Goto Beginning 13
†R Goto End 13
†R Goto Next Page 81
†R Goto Page 80
†R Goto Previous Page 81
†R Grow Window 74, 128
†R Incremental Search 41, 81
†R Indent for Comment 91
†R Indent for Lisp 93, 96
†R Indent Nested 98
†R Indent New Comment Line 92
†R Indent New Line 46, 87, 89, 96
†R Indent Region 47, 97
†R Indent Rigidly 47
†R Indent Sexp 97
†R Insert () 95
†R Insert Pagemark 81
†R Instant Extended Command 20
†R Join Next Page 81
†R Kbd Macro Query 126
†R Kill Comment 91
†R Kill Line 35
†R Kill Region 35
†R Kill Sentence 35, 45
†R Kill Sexp 35, 95
†R Kill Terminated Word 97
†R Kill Word 35, 43
†R Lowercase Region 50
†R Lowercase Word 49, 54
†R Mark Beginning 32
†R Mark Defun 32, 95
†R Mark End 32
†R Mark Page 32, 79
†R Mark Paragraph 32, 45
†R Mark Sexp 32, 95
†R Mark Whole Buffer 32
†R Mark Word 32, 44
†R Move Over) 95
†R Move to Screen Edge 72
†R New Window 13, 71, 161
†R Next Page 79
†R Next Screen 71
†R One Window 73
†R Open Line 15
†R Other Window 73
†R Previous Page 79
†R Previous Screen 71
†R Put Q-reg 39
†R Query Replace 84, 127
†R Quoted Insert 13
†R Re-execute Minibuffer 21, 84, 128
†R Reposition Window 7
†R Return to Superior 26
†R Reverse Search 41, 81

†R Save File 14, 55, 68
†R Scroll Other Window 73
†R Set Bounds Full 77, 83
†R Set Bounds Page 79
†R Set Bounds Region 77, 83
†R Set Comment Column 92
†R Set Fill Column 48, 111
†R Set Fill Prefix 48
†R Set/Pop Mark 31
†R Start Kbd Macro 125
†R Tab to Tab Stop 43, 47
†R Transpose Characters 13, 53
†R Transpose Lines 54
†R Transpose Regions 54
†R Transpose Sexps 95
†R Transpose Words 44
†R Two Windows 73
†R Un-kill 37
†R Un kill Pop 37
†R Underline Region 51
†R Underline Word 51
†R Universal Argument 17
†R Up Comment Line 92
†R Up Real Line 13
†R Upcase Digit 54
†R Uppercase Initial 49, 54
†R Uppercase Region 31, 50
†R Uppercase Word 49, 54
†R View Two Windows 74
†R Visit File 14, 55
†R Visit in Other Window 74
†R Widen Bounds 81